More about tibbles

In this reading, you will learn about tibbles, which are a super useful tool for organizing data in R. You will get a review of what tibbles are, how they differ from standard data frames, and how to create them in R.

**Tibbles**



Tibbles are a little different from standard data frames. A data frame is a collection of columns, like a spreadsheet or a SQL table. Tibbles are like streamlined data frames that are automatically set to pull up only the first 10 rows of a dataset, and only as many columns as can fit on the screen. This is really useful when you’re working with large sets of data. Unlike data frames, tibbles never change the names of your variables, or the data types of your inputs. Overall, you can make more changes to data frames, but tibbles are easier to use. The tibble package is part of the core tidyverse. So, if you’ve already installed the tidyverse, you have what you need to start working with tibbles.

**Creating tibbles**

Now, let’s go through an example of how to create a tibble in R. You can use the pre-loaded *diamonds* dataset that you’re familiar with from earlier videos. As a reminder, the *diamonds* dataset includes information about different diamond qualities, like carat, cut, color, clarity, and more.

You can load the dataset with the **data()** function using the following code:

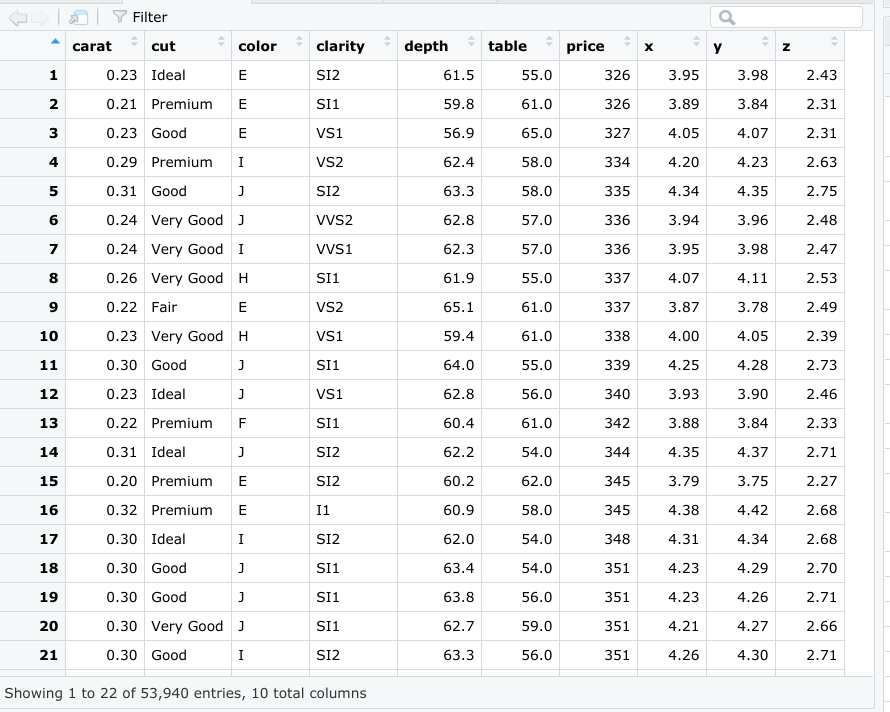
**library(tidyverse)**

**data(diamonds)**

Then, let’s add the data frame to our data viewer in RStudio with the **View()** function.

**View(diamonds)**

The dataset has 10 columns and thousands of rows. This image displays part of the data frame:

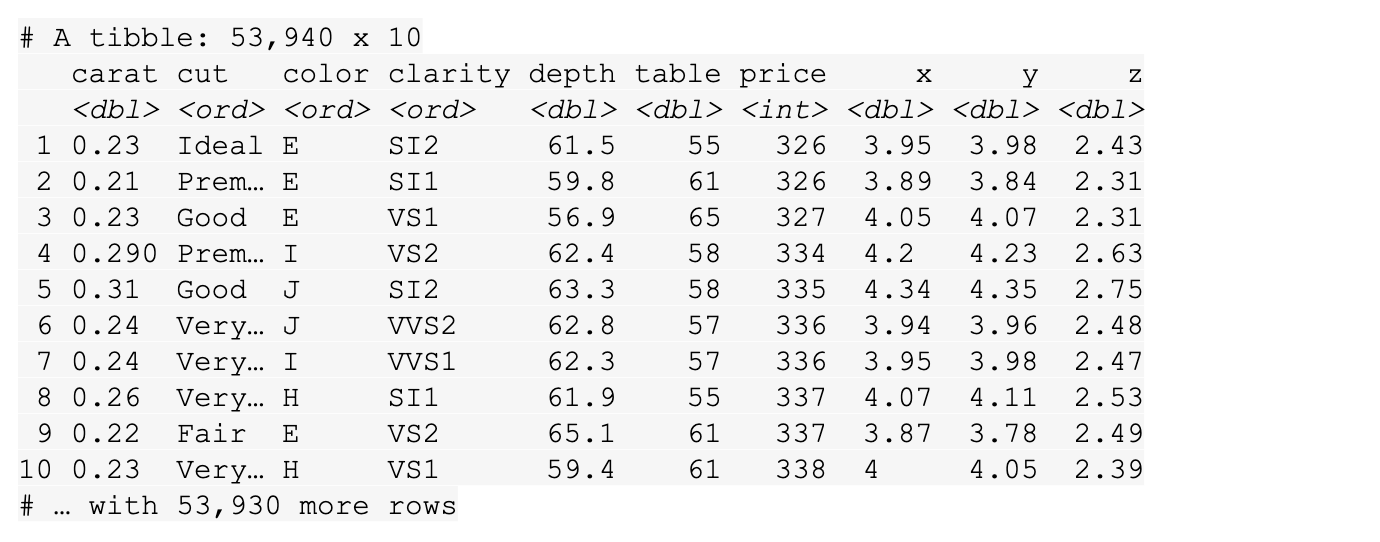


Now let’s create a tibble from the same dataset. You can create a tibble from existing data with the **as\_tibble()** function. Indicate what data you’d like to use in the parentheses of the function. In this case, you will put the word “diamonds."

**as\_tibble(diamonds)**

**Results**

When you run the function, you get a tibble of the *diamonds* dataset.



While RStudio’s built-in data frame tool returns thousands of rows in the *diamonds* dataset, the tibble only returns the first 10 rows in a neatly organized table. That makes it easier to view and print.

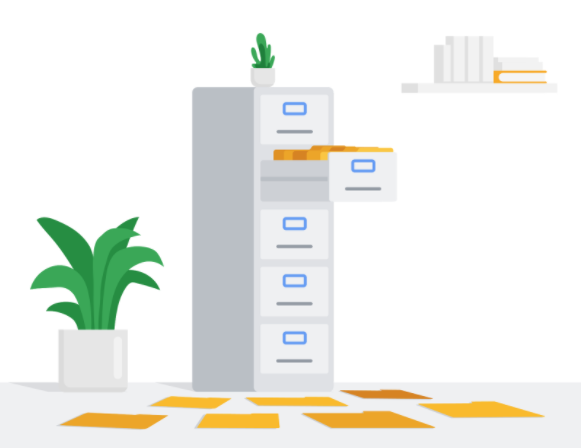
**Additional resources**

For more information on tibbles, check out the following resources:

* The entry for [Tibble](https://tibble.tidyverse.org/) in the tidyverse documentation summarizes what a tibble is and how it works in R code. If you want a quick overview of the essentials, this is the place to go.
* The [Tidy chapter](https://rstudio-education.github.io/tidyverse-cookbook/tidy.html) in "A Tidyverse Cookbook" is a great resource if you want to learn more about how to work with tibbles using R code. The chapter explores a variety of R functions that can help you create and transform tibbles to organize and tidy your data.

File-naming conventions

An important part of cleaning data is making sure that all of your files are accurately named. Although individual preferences will vary a bit, most analysts generally agree that file names should be accurate, consistent, and easy to read. This reading provides some general guidelines for you to follow when naming or renaming your data files.



**What’s in a (file)name?**

When you first start working with R (or any other programming language, analysis tool, or platform, for that matter), you or your company should establish naming conventions for your files. This helps ensure that anyone reviewing your analysis–yourself included–can quickly and easily find what they need. Next are some helpful “do’s” and “don’ts” to keep in mind when naming your files.

**Do**

* Keep your filenames to a reasonable length
* Use underscores and hyphens for readability
* Start or end your filename with a letter or number
* Use a standard date format when applicable; example: YYYY-MM-DD
* Use filenames for related files that work well with default ordering; example: in chronological order, or logical order using numbers first

| **Examples of good filenames** |
| --- |
| 2020-04-10\_march-attendance.R |
| 2021\_03\_20\_new\_customer\_ids.csv |
| 01\_data-sales.html |
| 02\_data-sales.html |

**Don't**

* Use unnecessary additional characters in filenames
* Use spaces or “illegal” characters; examples: &, %, #, <, or >
* Start or end your filename with a symbol
* Use incomplete or inconsistent date formats; example: M-D-YY
* Use filenames for related files that do not work well with default ordering; examples: a random system of numbers or date formats, or using letters first

| **Examples of filenames to avoid** |
| --- |
| 4102020marchattendance<workinprogress>.R |
| \_20210320\*newcustomeridsforfebonly.csv |
| firstfile\_for\_datasales/1-25-2020.html |
| secondfile\_for\_datasales/2-5-2020.html |

**Additional resources**

These resources include more info about some of the file naming standards discussed here, and provide additional insights into best practices.

* [**How to name files**](https://speakerdeck.com/jennybc/how-to-name-files): this resource from Speaker Deck is a playful take on file naming. It includes several slides with tips and examples for how to accurately name lots of different types of files. You will learn why filenames should be both machine readable and human readable.
* [**File naming and structure**](https://libguides.princeton.edu/c.php?g=102546&p=930626#:~:text=File%20naming%20best%20practices%3A&text=File%20names%20should%20be%20short,date%20format%20ISO%208601%3A%20YYYYMMDD): this resource from the Princeton University Library provides an easy-to-scan list of best practices, considerations, and examples for developing file naming conventions.

More on R operators

You might remember that an **operator** is a symbol that identifies the type of operation or calculation to be performed in a formula. In an earlier video, you learned how to use the assignment and arithmetic operators to assign variables and perform calculations. In this reading, you will review a detailed summary of the main types of operators in R, and learn how to use specific operators in R code.

**Operators**

 In R, there are four main types of operators:

1. Arithmetic
2. Relational
3. Logical
4. Assignment

Review the specific operators in each category and check out some examples of how to use them in R code.

**Arithmetic operators**

**Arithmetic operators** let you perform basic math operations like addition, subtraction, multiplication, and division.

The table below summarizes the different arithmetic operators in R. The examples used in the table are based on the creation of two variables: : *x* equals 2 and *y* equals 5. Note that you use the assignment operator to store these values:

**x <- 2**

**y <- 5**

| **Operator** | **Description** | **Example Code** | **Result/ Output** |
| --- | --- | --- | --- |
| + | Addition | x + y | [1] 7 |
| - | Subtraction | x - y | [1] -3 |
| \* | Multiplication | x \* y | [1] 10 |
| / | Division | x / y | [1] 0.4 |
| %% | Modulus (returns the remainder after division) | y %% x | [1] 1 |
| %/% | Integer division (returns an integer value after division) | y%/% x | [1] 2 |
| ^ | Exponent | y ^ x | [1]25 |

**Relational operators**

**Relational operators,** also known as comparators, allow you to compare values. Relational operators identify how one R object relates to another—like whether an object is less than, equal to, or greater than another object. The output for relational operators is either TRUE or FALSE (which is a logical data type, or boolean).

The table below summarizes the six relational operators in R. The examples used in the table are based on the creation of two variables: *x* equals 2 and *y* equals 5. Note that you use the assignment operator to store these values.

**x <- 2**

**y <- 5**

If you perform calculations with each operator, you get the following results. In this case, the output is boolean: TRUE or FALSE. Note that the [1] that appears before each output is used to represent how output is displayed in RStudio.

| **Operator** | **Description** | **Example Code** | **Result/Output** |
| --- | --- | --- | --- |
| < | Less than | x < y | [1] TRUE |
| > | Greater than | x > y | [1] FALSE |
| <= | Less than or equal to | x < = 2 | [1] TRUE |
| >= | Greater than or equal to | y >= 10 | [1] FALSE |
| == | Equal to | y == 5 | [1] TRUE |
| != | Not equal to | x != 2 | [1] FALSE |

**Logical operators**

**Logical operators** allow you to combine logical values. Logical operators return a logical data type or boolean (TRUE or FALSE)**.** You encountered logical operators in an earlier reading, [Logical operators and conditional statements](https://www.coursera.org/learn/data-analysis-r/supplement/I39VT/logical-operators-and-conditional-statements), but here is a quick refresher.

The table below summarizes the logical operators in R.

| **Operator** | **Description** |
| --- | --- |
| & | Element-wise logical AND |
| && | Logical AND |
| | | Element-wise logical OR |
| || | Logical OR |
| ! | Logical NOT |

Next, check out some examples of how logical operators work in R code.

**Element-wise logical AND (&) and OR (|)**

You can illustrate logical AND (&) and OR (|) by comparing numerical values. Create a variable *x* that is equal to 10.

**x <- 10**

The AND operator returns TRUE only if *both* individual values are TRUE.

**x > 2 & x < 12**

[1] TRUE

10 is greater than 2 *and* 10 is less than 12. So, the operation evaluates to **TRUE**.

The OR operator (|) works in a similar way to the AND operator (&). The main difference is that just *one* of the values of the OR operation needs to be TRUE for the entire OR operation to evaluate to TRUE. Only if *both* values are FALSE will the entire OR operation evaluate to **FALSE**.

Now try an example with the same variable **(x <- 10)**:

**x > 2 | x < 8**

**[1] TRUE**

10 is greater than 2, but 10 is not less than 8. But since at least one of the values (10>2) is TRUE, the OR operation evaluates to **TRUE**.

**Logical AND (&&)  and OR (||)**

The main difference between element-wise logical operators (&, |) and logical operators (&&, ||) is the way they apply to operations with vectors. The operations with double signs, AND (&&) and logical OR (||), only examine the *first* element of each vector. The operations with single signs, AND (&) and OR (|), examine all the elements of each vector.

For example, imagine you are working with two vectors that each contain three elements: **c(3, 5, 7)** and **c(2, 4, 6)**. The element-wise logical AND (&) will compare the first element of the first vector with the first element of the second vector (3&2), the second element with the second element (5&4), and the third element with the third element (7&6).

Now check out this example in R code.

First, create two variables, *x* and *y*, to store the two vectors:

**x <- c(3, 5, 7)**

**y <- c(2, 4, 6)**

Then run the code with a single ampersand (&). The output is boolean (TRUE or FALSE).

**x < 5 & y < 5**

**[1]  TRUE FALSE FALSE**

When you compare each element of the two vectors, the output is **TRUE, FALSE, FALSE**. The first element of both *x* (3) and *y* (2) is less than 5, so this is TRUE. The second element of x is *not* less than 5 (it’s equal to 5) but the second element of y is less than 5, so this is FALSE (because you used AND). The third element of both x and y is not less than 5, so this is also FALSE.

Now, run the same operation using the double ampersand (&&):

**x < 5 && y < 5**

**[1] TRUE**

In this case, R only compares the *first* elements of each vector: 3 and 2. So, the output is **TRUE** because 3 and 2 are both less than 5.

Depending on the type of work you do, you might make use of single sign operators more often than double sign operators. But it is helpful to know how all of the operators work regardless.

**Logical NOT (!)**

The NOT operator simply negates the logical value, and evaluates to its opposite. In R, zero is considered FALSE and all non-zero numbers are considered TRUE.

For example, apply the NOT operator to your variable **(x <- 10)**:

**!(x < 15)**

**[1] FALSE**

The NOT operation evaluates to **FALSE** because it takes the opposite logical value of the statement **x < 15**, which is TRUE (10 is less than 15).

**Assignment operators**

**Assignment operators** let you assign values to variables.

In many scripting programming languages you can just use the equal sign (=) to assign a variable. For R, the best practice is to use the arrow assignment (<-). Technically, the single arrow assignment can be used in the left or right direction. But the rightward assignment is not generally used in R code.

You can also use the double arrow assignment, known as a scoping assignment. But the scoping assignment is for advanced R users, so you won’t learn about it in this reading.

The table below summarizes the assignment operators and example code in R. Notice that the output for each variable is its assigned value.

| **Operator** | **Description** | **Example Code (after the sample code below, typing x will generate the output in the next column)** | **Result/ Output** |
| --- | --- | --- | --- |
| <- | Leftwards assignment | x <- 2 | [1] 2 |
| <<- | Leftwards assignment | x <<- 7 | [1] 7 |
| = | Leftwards assignment | x = 9 | [1] 9 |
| -> | Rightwards assignment | 11 -> x | [1] 11 |
| ->> | Rightwards assignment | 21 ->> x | [1] 21 |

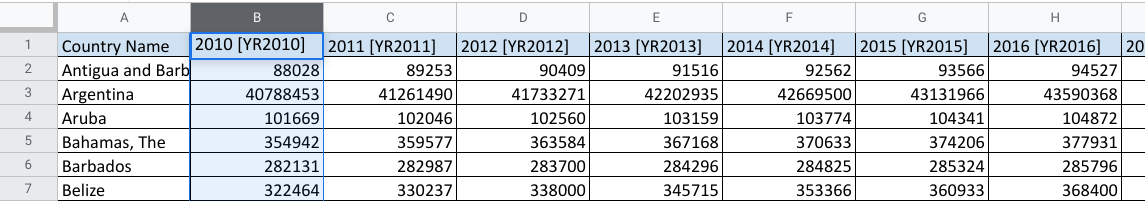
The operators you learned about in this reading are a great foundation for using operators in R.

**Additional resource**

Check out the article about [R Operators](https://r-coder.com/operators-r/#Assignment_operators_in_R) on the R Coder website for a comprehensive guide to the different types of operators in R. The article includes lots of useful coding examples, and information about miscellaneous operators, the infix operator, and the pipe operator.

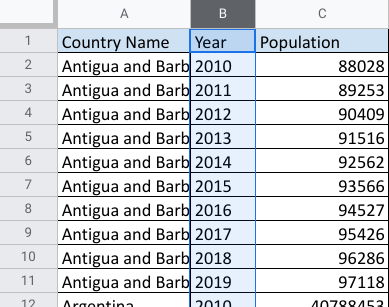
Wide to long with tidyr

When organizing or tidying your data using R, you might need to convert wide data to long data or long to wide. Recall that this is what data in a wide format looks like in a spreadsheet:



**Wide data** has observations across several columns. Each column contains data from a different condition of the variable. In this example, different years.

Now check out the same data in a long format:



And, to review what you already learned about the difference, **long data** has all the observations in a single column, and variables in separate columns.

**The pivot\_longer and pivot\_wider functions**



There are compelling reasons to use both formats. But as an analyst, it is important to know how to tidy data when you need to. In R, you may have a data frame in a wide format that has several variables and conditions for each variable. It might feel a bit messy.

That’s where pivot\_longer()comes in. As part of the tidyr package, you can use this R function to lengthen the data in a data frame by increasing the number of rows and decreasing the number of columns. Similarly, if you want to convert your data to have more columns and fewer rows, you would use the pivot\_wider() function.

**Additional resources**

To learn more about these two functions and how to apply them in your R programming, check out these resources:

* [**Pivoting**](https://tidyr.tidyverse.org/articles/pivot.html): Consider this a starting point for tidying data through wide and long conversions. This web page is taken directly from tidyr package information at [**tidyverse.org**](https://www.tidyverse.org/). It explores the components of the pivot\_longer and pivot\_wider functions using specific details, examples, and definitions.
* [**CleanItUp 5: R-Ladies Sydney: Wide to Long to Wide to…PIVOT**](https://rladiessydney.org/courses/ryouwithme/02-cleanitup-5/): This resource gives you additional details about the pivot\_longer and pivot\_wider functions. The examples provided use interesting datasets to illustrate how to convert data from wide to long and back to wide.
* [**Plotting multiple variables**](https://scc.ms.unimelb.edu.au/resources-list/simple-r-scripts-for-analysis/r-scripts)[**:**](https://www.datamentor.io/r-programming/saving-plot/) This resource explains how to visualize wide and long data, with ggplot2 to help tidy it. The focus is on using pivot\_longer to restructure data and make similar plots of a number of variables at once. You can apply what you learn from the other resources here for a broader understanding of the pivot functions.

Working with biased data

Every data analyst will encounter an element of bias at some point in the data analysis process. That’s why it’s so important to understand how to identify and manage biased data whenever possible. You might recall we explored bias in detail in Course 3 of this program. In this reading, you will read a real-life example of an analyst who discovered bias in their data, and learn how they used R to address it.

**Addressing biased data with R**



This scenario was shared by a quantitative analyst who collects data from people all over the world. They explain how they discovered bias in their data, and how they used R to address it:

“I work on a team that collects survey-like data. One of the tasks my team does is called a side-by-side comparison. For example, we might show users two ads side-by-side at the same time. In our survey, we ask which of the two ads they prefer. In one case, after many iterations, we were seeing consistent bias in favor of the first item. There was also a measurable decrease in the preference for an item if we swapped its position to second.

So we decided to add randomization to the position of the ads using R. We wanted to make sure that the items appeared in the first and second positions with similar frequencies. We used sample() to inject a randomization element into our R programming. In R, the sample() function allows you to take a random sample of elements from a data set. Adding this piece of code shuffled the rows in our data set randomly. So when we presented the ads to users, the positions of the ads were now random and controlled for bias. This made the survey more effective and the data more reliable.”

**Key takeaways**

The sample() function is just one of many functions and methods in R that you can use to address bias in your data. Depending on the kind of analysis you are conducting, you might need to incorporate some advanced processes in your programming. Although this program won’t cover those kinds of processes in detail, you will likely learn more about them as you get more experience in the data analytics field.

To learn more about bias and data ethics, check out these resources:

* [**Bias function:**](https://www.rdocumentation.org/packages/SimDesign/versions/2.2/topics/bias) This web page is a good starting point to learn about how the bias function in R can help you identify and manage bias in your analysis.
* [**Data Science Ethics**](https://datasciencebox.org/02-ethics.html): This online course provides slides, videos, and exercises to help you learn more about ethics in the world of data analytics. It includes information about data privacy, misrepresentation in data, and applying ethics to your visualizations.