A beginner's guide to R and RStudio

Beyond Spreadsheets with R

Dr. Jonathan Carroll

Sample Chapter

MANNING

Beyond Spreadsheets with R

by Dr. Jonathan Carroll

Chapter 3

brief contents

1 ■ Introducing data and the R language 1
2 ■ Getting to know R data types 26
3 ■ Making new data values 53
4 ■ Understanding the tools you'll use: Functions 67
5 ■ Combining data values 106
6 ■ Selecting data values 139
7 ■ Doing things with lots of data 182
8 ■ Doing things conditionally: Control structures 213
9 ■ Visualizing data: Plotting 235
10 ■ Doing more with your data with extensions 281

3 Making new data values

This chapter covers

- Performing operations between two or more data values
- Comparing values of the same or different type
- How R changes the data type as it needs to

You have your data values, but there's a good chance you'll want to do something with them, like add or multiply. It's time to go back to basics and see how R deals with combining data. Thankfully, R is a readable language for things like this, and with any luck you'll be able to code up what you're trying to do with the operators you expect. Follow along with this chapter in the Console and try some values yourself to see if you get the results you expect.

3.1 Basic mathematics

The simplest thing you might want to do to two values is add them. No surprises here, the + symbol (*operator*) between two values will do just that, the same as you would on a calculator:

```
2 + 2
#> [1] 4
```

The same goes for subtraction:

```
4 - 3
#> [1] 1
```

Multiplication in programming tends to use the *asterisk* (`*`) rather than a multiply sign (technically, ×, but commonly seen as an `x`):

```
3 * 6
#> [1] 18
```

Division uses a slash (`/`) like you might use in writing a fraction:

```
12 / 4
#> [1] 3
```

One that might not be so obvious is the *exponentiation* operator, which raises a number to a power. This one is used, for example, when you need to square a value. There are two options, though in reality they're different ways of writing the same thing. Here's the first:

```
2 ^ 10
#> [1] 1024
```

The second is much less common and may be confused with multiplication on too fast a skim-read,

```
2 ** 10
#> [1] 1024
```

Notice that no parentheses (`()`) were needed to group together the digits `1` and `0` into a `10`—R does some guesswork behind the scenes to interpret what you mean, and treats this as the value `10` rather than an invalid expression. The spaces between values and operators are also cleverly interpreted; you could remove them and write `3+2` or `3^2`, but for the sake of clarity, it's often best to space things out a little. The less-common variant `**` does, however, require that there be *no* spaces between the two asterisks.

Dates are a special type in R (recall section 2.1.4), and you can now see why. Although you can certainly subtract dates to calculate a period of time, which R interprets sensibly for suitably encoded values

```
as.POSIXct(x = "2016-12-31") - as.POSIXct(x = "2016-01-01")
#> Time difference of 365 days
```

adding dates doesn't make sense:

```
as.POSIXct(x = "2016-12-31") + as.POSIXct(x = "2016-01-01")

#> Error: binary '+' is not defined for "POSIXt" objects
```

Somewhat related, R won't let you perform operations between incompatible types for a given operator. You might understand the intention of the following perfectly well:

```
2 + "2"

#> Error: non-numeric argument to binary operator
```

But R won't take the leap of faith in assuming that you mean to add these as numbers. A spreadsheet may have the same apprehensions about adding a number to a string, an example of which is shown in figure 3.1.

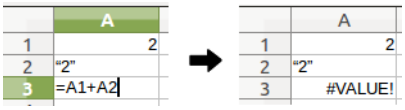

Figure 3.1 Attempting to add a number to a string in a spreadsheet, resulting in an error

Spreadsheets may, however, be more flexible when attempting this within a formula, as in figure 3. 2.

Figure 3.2 Attempting to add a number to a string within a formula in a spreadsheet, which doesn't result in an error

R is less fussy over the difference between numeric and integer types, and will allow you to work with these interchangeably if you ask. It returns the result in the most general type possible (in this case, `numeric`, which is more general than `integer`):

```
numPlusInt <- 7 + 3L
str(object = numPlusInt)
#>  num 10
```

It should come as no surprise then that certain combinations are out of the question:

```
"7" - "4"

#> Error: non-numeric argument to binary operator

"7" + "4"

#> Error: non-numeric argument to binary operator
```

Others may surprisingly work just fine:

```
as.POSIXct(x = "2016-12-31") + 1
#> [1] "2016-12-31 00:00:01 ACDT"
```

Remember that the time zone shown here (Australian Central Daylight Time, ACDT) is likely to be different from your own output in your time zone.

Here R has treated the `1` as some number of *seconds* to add to the date-time object on the left side of the `+` operator. To add a whole day, then you need to add 24 hours, each of which has 60 minutes, each of which has 60 seconds:

```
as.POSIXct(x = "2016-12-31") + 60*60*24
#> [1] "2017-01-01 ACDT"
```

Although you certainly mean to be multiplying integers, which you would specify with an L suffix, you can be lazy and multiply numeric values and get the same result.

Math with NA

Involving the missing data value `NA` typically leads to more missing data. Trying to add missing data to known data results in the total value going missing:

```
7 + NA
#> [1] NA
NA + 0
#> [1] NA
```

This can lead to some unwanted effects if you have a lot of data values and just one missing value, perhaps because it was converted from another type incorrectly. The `sum()` function, which calculates the sum of its inputs, has an option specifically for this scenario (as do several functions). If you try to take the sum of some values including an `NA` value

```
sum(3, 7, 0, 9, NA)
#> [1] NA
```

the result is missing. But if you tell this function to first remove any `NA` values with the option `na.rm = TRUE`

```
sum(3, 7, 0, 9, NA, na.rm = TRUE)
#> [1] 19
```

the `NA` value is removed before the sum is calculated.

It's best to be suspicious any time you may have `NA` values in your data.

3.2 Operator precedence

When multiple operations are acting together, R has rules that determine the order in which they will be evaluated, called *precedence*.

DEFINITION *Precedence* refers to the order or ranking of a group. In R, this refers to which operations will be performed before others.

For example, perhaps you've seen the "challenges" floating around social media, with "99% of people won't get this right" posted alongside a short and semi-ambiguous mathematical statement, such as

```
2 + 3 * 0 - 1 = ?
```

These should be no hassle for anyone who remembers their order of operations from school: the acronym PEMDAS stands for Parentheses, Exponents, Multiplication and Division, and Addition and Subtraction—the order in which these operations should be carried out. In the preceding case, given the lack of parentheses (`()`) or exponents (raised to a power), the next step is to perform the multiplication of `3` and `0`, which is `0`. This leaves the addition and subtraction steps, which don't depend on their order:

```
2 + 0 - 1 = 1
```

Nonetheless, the comments on said "challenge" will contain any number of answers and people fighting tooth and nail to defend their incorrect justifications.

R has a similar hierarchy for order of operations, but it includes some other operators we haven't discussed yet. Nonetheless, it's good practice to include parentheses where there is doubt about what the intended order of operations is, or to force a particular grouping. In the preceding case, writing this as

```
2 + (3 * 0) - 1
```

would help make it clearer that the `3` is multiplied by the `0` and the result of that takes part in the subsequent addition and subtraction. This becomes essential when you want to exponentiate to the result of some combination of multiple values:

```
2 ^ (5 * 2)
```

The parentheses dictate that this group needs to be evaluated first, before the exponentiation. This can significantly alter the way an expression is evaluated. Consider for a moment the following odd-looking expression:

```
x <- y <- 3 + 1
```

This is perfectly valid, and R will process it as it understands it (assign the result of `3 + 1` to `y`, and assign that result to `x`), producing the following:

```
x
#> [1] 4

y
#> [1] 4
```

Including some more parentheses, though,

```
x <- (y <- 3) + 1
```

you can change what the expression means (now assign `3` to `y`, then add `1`, and then assign that result to `x`) producing

```
x
#> [1] 4

y
#> [1] 3
```

The computer and R have no problems understanding what you've written (a certain way). Do your best to make sure that what you've written is what you mean.

3.3 String concatenation (joining)

You saw earlier that adding two strings (`"7" + "4"`) produced an error, because the `+` operator works for numbers (numeric or integer). We could envisage trying to "add" two words together, though, perhaps `"butter"` and `"fly"`. But we don't really mean *add*—we mean *join*, or more strictly, *concatenate*.

For these circumstances, there is a specific function to perform this operation, the `paste()` function, but it has a default of joining strings with a space between them (which is often what you want):

```
paste("butter", "fly")
#> [1] "butter fly"
```

This is the result of the default argument `sep = " "`, which specifies the separator to place between the inputs, with a default of using a space. You can change this to `sep = ""` to remove it entirely, or use the convenience function

```
paste0("butter", "fly")
#> [1] "butterfly"
```

which performs the same operation but with 0 spaces between inputs.

The `paste()` (and `paste0()`) function converts its input to `character` before pasting together, so you can use other types here too:

```
paste0("value", 31)
#> [1] "value31"
```

You can even use variables you've defined, or other `paste()` calls:

```
address_number <- 221
address_suffix <- "B"
address_street <- "Baker Street"

paste(
  paste0(
    address_number,
    address_suffix
  ),
  address_street
)
#> [1] "221B Baker Street"
```

Inputs to the innermost paste0()

Input to the outermost paste()

Joining NA values

This is one scenario in which a missing value can become non-missing, perhaps unexpectedly. By default, the inputs to `paste()` are converted to type `character` with the `as.character()` function. But that function preserves missing values, so these two are different results:

```
as.character(x = NA)
#> [1] NA
as.character(x = "NA")
#> [1] "NA"
```

A missing value

The string "NA"

You might expect then that `paste()` would produce a missing value when one of its inputs is `NA`, but it handles this smoothly:

```
paste("a missing value is denoted", NA)
#> [1] "a missing value is denoted NA"
```

This is noteworthy enough to receive special mention in the `help()` page for `paste()`: "Note that `paste()` coerces `NA_character_`, the character missing value, to 'NA' which may seem undesirable, e.g., when pasting two character vectors, or very desirable, e.g. in `paste('the value of p is ', p)`."

Enter some more values into the Console with combinations of these operators and make sure you're comfortable with them. In the next section, you'll see how to make comparisons between values.

3.4 Comparisons

The essence of scientific results boils down to comparing values. Are there fewer `y` than a decade ago? Has `z` grown this week? Is `j` faster than `k`? Are any of `m` significant effects (fitting some criteria)? If we didn't require comparisons between data, coding analyses would be fairly straightforward—always do this, then that, then the other thing. Because we do require comparisons, though, we must know how to tell R to compare values.

The result of a comparison will always be a *logical* value (recall from section 2.1.5): either `TRUE` or `FALSE` (or missing: `NA`). The simplest comparisons you can make are "Is `x` greater than `y`?" (`x > y`) and "Is `x` less than `y`?" (`x < y`), using the "greater than" and "less than" operators, `<` and `>`:

```
7 > 4
#> [1] TRUE

9 < 3
#> [1] FALSE
```

> **CAUTION** Recall the example in section 2.2.3, which accidentally included a space in `<-` and produced a comparison. Be very careful with spaces. Whitespace can (and should) be included around the comparison operators to help make the code clear. It's all too easy to miss the difference between `x< -3` and `x <- 3`.

You can also allow for the possibility of "Is `x` greater than or equal to `y`?" (`x >= y`) and its partner "Is `x` less than or equal to `y`?" (`x <= y`):

```
5 >= 6
#> [1] FALSE

3 <= 3
#> [1] TRUE
```

You can test whether two numbers are equal to each other with a "double equals" (`==`), which asks "Is `x` the same value as `y`?" (`x == y`). The opposite question can be asked also: "Is `x` a different value than `y`?" or "Does `x` not equal `y`?" (`x != y`):

```
3 == 3
#> [1] TRUE

7 != 4
#> [1] TRUE
```

Some care needs to be taken with these operators. They test whether two values appear to be the same, but not whether they are *precisely* the same. Even though you can specify an integer-like value as either an integer or a real number, these are

stored differently. They represent the same value, though, so == treats these as equal when testing:

```
5L == 5
#> [1] TRUE
```

If you truly want to test whether these are the same thing, the `identical()` function checks the types of the input values before declaring things identical:

```
identical(x = 5L, y = 5)
#> [1] FALSE
```

Comparisons between real values

Be very careful when comparing non-integer (real) numbers against each other. Recall from chapter 2 that computers store data in binary (as ones and zeroes). Because of this, there is a limitation to the precision with which decimal numbers can be stored. You may enter the value `0.3`, but the computer attempts to store as many digits (zeroes in this case) at the end of that as it can. In the same way that you can't really write `1/3 = 0.33333...` exactly without going on forever, computers can't store values that aren't exact powers of 2 without being off by a little.

You can see this in effect by requesting more digits in the output from a `print()` command:

```
print(x = 0.3, digits = 17)
#> [1] 0.29999999999999999
```

When mathematical operations take place, these extra or lacking bits also contribute. So although you may see a nice rounded value in your output, in terms of the digits the computer knows about, it may be slightly off from that, leading to unexpected results:

```
print(x = 0.1 + 0.2)
#> [1] 0.3
print(x = 0.1 + 0.2, digits = 17)
#> [1] 0.30000000000000004
```

This isn't unique to R.[1] Rather, it's inherent in any computer language. The safest way to avoid this issue is to never compare real numbers against each other if you need the answer to be exact:

```
0.1 + 0.2 > 0.3
#> [1] TRUE
```

The `!` in `!=` represents the notion of *not*, as in *not equal*. This can appear in several different places with the same effect:

```
3 != 4
#> [1] TRUE
```

[1] See the article "Floating Point Math" at http://0.30000000000000004.com.

```
!(3 == 4)
#> [1] TRUE

(! 3 == 4)
#> [1] TRUE
```

What if you try to compare to a missing value? The comparison of anything to the missing value `NA` is `NA`

```
3 > NA
#> [1] NA

7 == NA
#> [1] NA
```

even if the comparison involves a "not" operator:

```
5 != NA
#> [1] NA
```

Comparison between the two logical values can be summarized in a *truth table*, where the intersection of a row and a column shows the result of the comparison `row == column`, as shown in table 3.1. Notice that comparing anything to `NA` yields `NA`, including `NA` itself.

Table 3.1 Truth table for the equals operator (==)

==	TRUE	FALSE	NA
TRUE	TRUE	FALSE	NA
FALSE	FALSE	TRUE	NA
NA	NA	NA	NA

Comparing anything to `NULL` results in nothing, but a particular "type" of nothing—still a logical value, but one that is of length 0:

```
3 > NULL
#> logical(0)

TRUE == NULL
#> logical(0)

NA != NULL
#> logical(0)
```

There are other ways to compare logical values; you can also combine them with *and* (`&`) and *or* (`|`) operators. Let's make another truth table for the logical options, as shown in table 3.2 for *and*.

Table 3.2 Truth table for the *and* operator (`&`)

&	TRUE	FALSE	NA
TRUE	TRUE	FALSE	NA
FALSE	FALSE	FALSE	FALSE
NA	NA	FALSE	NA

The surprising result of table 3.2 is the combination of `NA` and `FALSE`, which yields `FALSE`. The help page for `` ?`&` `` explains: "`NA` is a valid logical object. Where a component of `x` or `y` is `NA`, the result will be `NA` if the outcome is ambiguous."

A similar table can be constructed for the *or* (`|`) operator, as shown in table 3.3. Again, comparison with `NA` doesn't necessarily lead to `NA` as long as the comparison is unambiguous.

Table 3.3 Truth table for the *or* operator (`|`)

|	TRUE	FALSE	NA
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NA
NA	TRUE	NA	NA

Individual comparisons

Although it may be tempting to think that you can write logical combinations the way you might read them, it's important to remember that only one comparison can be made at a time. To test whether some variable `x` was greater than `3`, you would write the following:

```
x <- 4
x > 3
#> [1] TRUE
```

To *also* test whether `x` was less than `5`, you might try adding in this condition as "also less than `5`":

```
x > 3 & < 5
#> Error: unexpected '<' in "x > 3 & <"
```

This fails because when R reads in this expression, it breaks it down into the appropriate levels of precedence (recall section 3.2), and `<` has a higher precedence (is evaluated earlier) than `&`; but here `<` has nothing to compare to, so the call fails.

Instead, you need to repeat a little and make the two comparisons explicit:

```
x > 3 & x < 5
#> [1] TRUE
```

Equivalent to (x > 3) & (x < 5)

There are also double forms of *and* (`&&`) and *not* (`||`) operators that evaluate, left to right, one element at a time, until a result is definitely reached. This becomes important and useful when you have collections of logical comparisons to test, but only require a single answer. You may need to make a thousand `&` comparisons to test whether all the values of two inputs are the same; but if they differ in the fourth pair then they're *definitely* not the same, so the rest of the comparisons can be skipped.

Testing whether a value is equal to NA

If you need to compare whether a value is `NA`, the handy built-in function `is.na()` returns `TRUE` if the input is `NA`, and `FALSE` otherwise:

```
is.na(x = 7)
#> [1] FALSE
is.na(x = NA)
#> [1] TRUE
```

3.5 Automatic conversion (coercion)

Occasionally, R will perform a conversion on your behalf (whether you wanted it to or not). Typically, this is a useful feature; to add an integer and a non-integer, you might try this:

```
3L + 2.5
#> [1] 5.5
```

The result can be seen by examining the structure:

```
str(object = 3L + 2.5)
#>  num 5.5
```

What has happened here? The C code underlying the `+` function has a few lines specifically for this scenario. If *one* (and only one) of the arguments (with the shorthand `+` notation, the left or right side) is of type `numeric` and the other is of type `integer`, the integer value is *coerced* to a numeric value, and then the two are added, producing a final numeric value. This is the result you want—converting an integer to a numeric value is essentially trivial: no rounding is required. Converting from a numeric value to an integer, however, has some potential complications, as you saw in section 2.3. If automatic coercion takes place, R will always coerce to the more general structure. The ordering from most specific to most general is shown in figure 3.3.

Figure 3.3 Ordering of coercion

R considers certain values to be equivalent to `TRUE` and others equivalent to `FALSE`. For numbers, the value `0` (or `0L`) converts to `FALSE`, whereas any other (positive or negative) number converts to `TRUE`. You can observe this by requesting the conversion explicitly:

```
as.logical(x = 0)
#> [1] FALSE

as.logical(x = 1)
#> [1] TRUE
```

Or compare with the approximate test `==`:

```
0 == FALSE
#> [1] TRUE

1 == TRUE
#> [1] TRUE
```

The automatic conversion the other way means you can add logical and real values together:

```
8 + TRUE
#> [1] 9
```

This becomes very handy for counting binary values:

```
sum(TRUE, FALSE, TRUE, TRUE)
#> [1] 3
```

Performing comparisons is when you are most likely to encounter automatic coercion. You may not expect the following to work at all, but it does:

```
"a" > 5
#> [1] TRUE
```

When R notices you're trying to compare two different types, it performs coercion to the most general type—in this case, `character` is more general (it's the most general) than `numeric`. What happens next depends on where you live; the strings/characters are compared by their place in the encoding scheme's table. This is language-dependent (which tends to depend on the *locale* your computer recognizes): this example is using (Australian) English with a `UTF-8` scheme (`en_AU.UTF-8`), for which letters come *after* numbers. The `help()` page for the comparison operators (for example, `?`>`) notes that in Estonian, *Z* comes between *S* and *T*,[2] so be aware when using this feature that your results may differ from someone else's.

[2] Read more about this in the Wikipedia article at http://mng.bz/5Yx8.

This is also the reason the wayward space in the assignment operation back in section 2.3 produced a result. Recall that you tried to assign the value `3` to the variable `a` but accidentally inserted a space inside `<-`, and so instead generated a logical value since `a` already held the value `"x"`:

```
a <- "x"
a < - 3
#> [1] FALSE
```

Here R sees a comparison between two different types, so R converts to the most general type (in this case, `character`) and checks the encoding scheme's table to see if `"x"` comes before or after `-3`. In my locale, letters come *after* numbers, so this comparison returns `FALSE`.

You could reasonably expect that the following is automatically coercing to numeric

```
"2" < "3"
#> [1] TRUE
```

which would produce the output as shown. A simple change shows that this isn't the case:

```
"2" < "13"
#> [1] FALSE
```

In these cases, no coercion is required, because both values are already of type `character`. They are again compared via their positions in the encoding table, and these are sorted numerically such that `"13"` (starting with a `"1"`) is before `"2"`.

3.6 Try it yourself

Daily temperatures are typically found in one of two scales: Celsius and Fahrenheit. It's useful to know how to convert between the two. If you have a temperature in Fahrenheit and want to convert it to Celsius, you can enter the Fahrenheit value into this expression to find the Celsius value:

```
Celsius <- (5 * Fahrenheit - 32) / 9
```

So, if you have

```
temp_F <- 88
```

you can calculate the temperature in Celsius using

```
temp_C <- 5L * (temp_F - 32L) / 9L
```

(the `L` specifications aren't essential, but are good practice). You can now examine what value this takes:

```
temp_C
#> [1] 31.11111
```

Going the other way, you can invert the expression:

```
Fahrenheit = (9 * Celsius / 5) + 32
```

So you should be able to obtain the original value again:

```
temp_F_recalculated <- (9L * temp_C / 5L) + 32L
temp_F_recalculated
#> [1] 88
```

You can reassure yourself that this is in fact the same value:

```
temp_F == temp_F_recalculated
#> [1] TRUE
```

Evaluate the preceding conversions (or better yet, write your own) and convert 0°C to Fahrenheit. Convert 100°F to Celsius. Convert –40° from one scale to the other.

Terminology

- *Operator*—A symbol representing an operation to be performed on one or more values: for example, `+`
- *Precedence*—The order in which operations will be evaluated
- *Expression*—A command for R to interpret; performing an operation on 0 or more data values

Summary

- You can use R as a calculator more or less as you'd expect.
- Certain types of data can't be added/subtracted.
- Operators have different precedence, which can be overridden with parentheses.
- Strings can be combined using `paste()`.
- Coercion will result in data being converted to the most general type.
- `Character` is the most general type of data.
- Text can be compared to numbers because the latter will be coerced to `character`.
- Comparisons between real numbers is dangerous due to rounding differences.
- Comparing `NA` to anything produces `NA`, even `NA` itself.
- Strings can be compared, but doing so depends on your computer's settings, usually dependent on where you live.

R/DATA SCIENCE

Beyond Spreadsheets with R

Dr. Jonathan Carroll

Spreadsheets are powerful tools for many tasks, but if you need to interpret, interrogate, and present data, they can feel like the wrong tools for the task. That's when R programming is the way to go. The R programming language provides a comfortable environment to properly handle all types of data. And within the open source RStudio development suite, you have at your fingertips easy-to-use ways to simplify complex manipulations and create reproducible processes for analysis and reporting.

With **Beyond Spreadsheets with R** you'll learn how to go from raw data to meaningful insights using R and RStudio. Each carefully crafted chapter covers a unique way to wrangle data, from understanding individual values to interacting with complex collections of data, including data you scrape from the web. You'll build on simple programming techniques like loops and conditionals to create your own custom functions. You'll come away with a toolkit of strategies for analyzing and visualizing data of all sorts.

What's Inside

- How to start programming with R and RStudio
- Understanding and implementing important R structures and operators
- Installing and working with R packages
- Tidying, refining, and plotting your data

If you're comfortable writing formulas in Excel, you're ready for this book.

Jonathan Carroll is a data science consultant providing R programming services. He holds a PhD in theoretical physics.

"A useful guide to facilitate graduating from spreadsheets to more serious data wrangling with R."
—John D. Lewis, DDN

"An excellent book to help you understand how stored data can be used."
—Hilde Van Gysel
Trebol Engineering

"A great introduction to a data science programming language. Makes you want to learn more!"
—Jenice Tom, CVS Health

"Handy to have when your data spreads beyond a spreadsheet."
—Danil Mironov, Luxoft Poland

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/beyond-spreadsheets-with-r

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]

ISBN-13: 978-1-61729-459-4
ISBN-10: 1-61729-459-4
54999
9 781617 294594