Vectors

Chapter 2

Stats 20: Introduction to Statistical Programming with R

UCLA

Contents

Acknowledgements: Vivian Lew and Juana Sanchez

Learning Objectives

After studying this chapter, you should be able to:

- Combine vectors with the `c()` function.
- Understand the distinction between numeric, character, and logical vectors.
- Extract values from a vector using subsetting.
- Compute vector arithmetic in R.
- Understand how R uses recycling in vector operations.
- Understand how R uses vectorization.
- Understand that R approximates numbers to identify and be aware of rounding errors.

1 The Essentials

1.1 Basic Definitions

The most fundamental object in R is a **vector**, which is an ordered collection of values. The entries of a vector are also called *elements* or *components*. Single values (or **scalars**) are actually just vectors with a single element.

The possible values contained in a vector can be of several basic data types, also known as (storage) **modes**: numeric, character, or logical.

- **Numeric** values are numbers (decimals).
- **Character** values (also called **strings**) are letters, words, or symbols. Character values are always contained in quotation marks `""`.
- **Logical** values are either `TRUE` or `FALSE` (must be in all caps), representing true and false values in formal logical statements.

Note: The (capital) letters `T` and `F` are technically valid shorthand for `TRUE` and `FALSE`, respectively, but you should *never* use them.

The `c()` function is used to collect values into a vector. The `c` stands for concatenate or combine. Here are a few examples:

```
c(1, 1, 2, 3, 5, 8, 13) # This is a numeric vector
```

```
[1]  1  1  2  3  5  8 13
```

```
fib <- c(1, 1, 2, 3, 5, 8, 13) # Assign the vector to a named object
fib
```

```
[1]  1  1  2  3  5  8 13
```

```
parks <- c("Leslie", "April", "Ron", "Tom", "Donna", "Jerry") # This is a character vector
parks
```

```
[1] "Leslie" "April"  "Ron"    "Tom"    "Donna"  "Jerry"
```

```
true_dat <- c(TRUE, FALSE, TRUE, T, F) # This is a logical vector
true_dat
```

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

The `c()` function can also concatenate vectors together by inputting vectors instead of single values.

```r
c(c(1, 2), c(3, 4, 5)) # Can concatenate multiple vectors together
```

```
[1] 1 2 3 4 5
```

1.2 The Length of a Vector

The **length** of a vector is the number of elements in the vector. The **`length()`** function inputs a vector and outputs the length of the vector.

```r
length(4) # A scalar/number is a vector of length 1
```

```
[1] 1
```

```r
length(fib)
```

```
[1] 7
```

```r
length(parks)
```

```
[1] 6
```

```r
length(true_dat)
```

```
[1] 5
```

1.3 The Mode Hierarchy

In the examples above, we have created separate numeric, character, and logical vectors, where all the values in each vector are of the same type. A natural question is whether we can create a vector with mixed types.

It turns out that the answer is *no*: Due to how R (internally) stores vectors, *every value in a vector must have the same type.*

The **`mode()`** function inputs an object and outputs the type (or mode) of the object. This is a general function that can be applied to all objects, not just vectors.

```r
mode(fib)
```

```
[1] "numeric"
```

```r
mode(parks)
```

```
[1] "character"
```

```r
mode(true_dat)
```

```
[1] "logical"
```

When values of different types are concatenated into a single vector, the values are **coerced** into a single type.

Question: What is the output for the following commands?

- `mode(c(fib, parks))`
- `mode(c(fib, true_dat))`
- `mode(c(parks, true_dat))`
- `mode(c(fib, parks, true_dat))`

These questions highlight the **mode hierarchy**:

$$\text{logical} < \text{numeric} < \text{character}$$

That is:

- Combining logical and numeric vectors will result in a numeric vector.
- Combining numeric and character vectors will result in a character vector.
- Combining logical and character vectors will result in a character vector.
- Combining logical, numeric, and character vectors will result in a character vector.

Note: When logical values are coerced into numeric values, `TRUE` becomes `1` and `FALSE` becomes `0`.

The reason why knowing the types of our R objects is important is because we want to apply functions to the objects in order to describe, visualize, or do analysis on them. Just like in mathematics, functions in R are all about input and output. Functions will expect inputs (arguments) in a certain form and will give outputs in other forms, such as other R objects (vectors, matrices, data frames, etc.) or plots. In addition, some functions will change their output depending on the input.

As you become more familiar with R, it is important to know what type your objects are and what functions are available to you for working with those objects.

2 Sequences and Repeated Patterns

R has some handy built-in functions for creating vectors of sequential or repeated values. One common use of sequences in statistics is to generate the category labels (or **levels**) for a designed experiment.

2.1 The `seq()` Function

The **`seq()`** function creates a sequence of evenly spaced numbers with specified start and end values. The start and end values determine whether the sequence is increasing or decreasing. The first argument is the `from` or starting value, and the second argument is the `to` or end value. By default, the optional argument by is set to `by = 1`, which means the numbers in the sequence are incrementally increased by 1.

```
seq(0, 5) # numbers increase by 1
```

```
[1] 0 1 2 3 4 5
```

```
seq(0, 10, by = 2) # numbers now increase by 2
```

```
[1]  0  2  4  6  8 10
```

The `seq()` function can make decreasing sequences by specifying the `from` argument to be greater than the `to` argument. By default, the `by` argument will automatically change to `by = -1`.

```
seq(5, 0) # seq() can also make decreasing sequences
```

```
[1] 5 4 3 2 1 0
```

```
seq(10, 0, by = -3) # numbers now decrease by 3
```

```
[1] 10  7  4  1
```

Notice that `seq(10, 0, by = -3)` stops at the smallest number in the sequence greater than the `to` argument.

To obtain a sequence of numbers of a given length, use the optional `length` (or `length.out`) argument. The incremental increase (or decrease) will be calculated automatically.

```
seq(0, 1, length = 11)
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

We could also specify the increment and length instead of providing the end value.

```r
seq(10, 55, length = 10)
```

```
 [1] 10 15 20 25 30 35 40 45 50 55
```

```r
seq(10, by = 5, length = 10) # The same sequence
```

```
 [1] 10 15 20 25 30 35 40 45 50 55
```

2.1.1 Shorthands for Common Sequences

R has several shorthands for common sequences. We will discuss the colon `:` operator, `seq_len()`, and `seq_along()`.

2.1.1.1 The Colon `:` Operator The colon `:` operator is a shorthand for the default `seq()` with unit increment (i.e., `by = 1` or `-1`).

```r
-2:5 # same as seq(-2, 5)
```

```
[1] -2 -1  0  1  2  3  4  5
```

```r
pi:10 # same as seq(pi, 10)
```

```
[1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593
```

Caution: The colon `:` operator takes precedence over multiplication and subtraction in the order of operations, but it does not take precedence over exponents. It is always recommended to use parentheses to make the order of operations explicit.

```r
n <- 5
1:n - 1
```

```
[1] 0 1 2 3 4
```

```r
1:(n - 1)
```

```
[1] 1 2 3 4
```

2.1.1.2 The `seq_len()` Function The `seq_len()` function inputs a single `length.out` argument and generates the sequence of integers `1, 2, ..., length.out` unless `length.out = 0`, when it generates `integer(0)`.

```r
seq_len(8)
```

```
[1] 1 2 3 4 5 6 7 8
```

```r
seq_len(10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq_len(0)
```

```
integer(0)
```

Notice that the output of `1:n` and `seq_len(length.out = n)` are the same for positive integers `n`. However, if `n = 0`, then `seq_len(n)` will generate `integer(0)`, whereas `1:n` will produce the vector `1, 0`, which is often not the intended behavior when using the `1:n` notation (especially when used inside of functions). In addition, `seq_len()` does not allow for negative inputs.

```r
seq_len(-5)
```

```
Error in seq_len(-5): argument must be coercible to non-negative integer
```

Using `seq_len(n)` rather than the shorter `1:n` helps prevent unexpected results if `n` is incorrectly specified. When creating an integer sequence of possibly variable length, the `seq_len()` notation is recommended best practice over the colon operator `:`.

2.1.1.3 The `seq_along()` Function The `seq_along()` function inputs a single `along.with` argument and generates the sequence of integers `1, 2, ..., length(along.with)`.

```r
seq_along(100)
```

```
[1] 1
```

```r
seq_along(c(1, 3, 5, 7, 9))
```

```
[1] 1 2 3 4 5
```

```r
seq_along(c("friends", "waffles", "work"))
```

```
[1] 1 2 3
```

The `seq_along()` function can be useful for generating a sequence of indices for the input vector, which will be helpful when writing loops (as we will see in a later chapter).

2.2 The `rep()` Function

The `rep()` function creates a vector of repeated values. The first argument, generically called `x`, is the vector of values we want to repeat. The second argument is the `times` argument that specifies how many times we want to repeat the values in the `x` vector.

The `times` argument can be a single value (repeats the whole vector) or a vector of values (repeats each individual value separately). If the length of the `times` vector is greater than 1, the vector needs to have the same length as the `x` vector. Each element of `times` correponds to the number of times to repeat the corresponding element in `x`.

```r
rep(3, 10) # Repeat the value 3, 10 times
```

```
 [1] 3 3 3 3 3 3 3 3 3 3
```

```r
rep(c(1, 2), 5) # Repeat the vector c(1,2), 5 times
```

```
 [1] 1 2 1 2 1 2 1 2 1 2
```

```r
rep(c(1, 2), c(4, 3)) # Repeat the value 1, 4 times, and the value 2, 3 times
```

```
[1] 1 1 1 1 2 2 2
```

```r
rep(c(5, 3, 1), c(1, 3, 5)) # Repeat c(5,3,1), c(1,3,5) times
```

```
[1] 5 3 3 3 1 1 1 1 1
```

Question: How is `rep(c(1, 2), 5)` different from `rep(c(1, 2), c(5, 5))`?

Question: Why does `rep(c(5, 3, 1), c(1, 3))` give an error?

We can also combine `seq()` and `rep()` to construct more interesting patterns.

```
rep(seq(2, 20, by = 2), 2)
```

```
 [1]  2  4  6  8 10 12 14 16 18 20  2  4  6  8 10 12 14 16 18 20
```

```
rep(seq(2, 20, by = 2), rep(2, 10))
```

```
 [1]  2  2  4  4  6  6  8  8 10 10 12 12 14 14 16 16 18 18 20 20
```

Note: The `rep()` function works with vectors of any mode, including character and logical vectors. This is particularly useful for creating vectors that represents categorical variables.

```
rep(c("long", "short"), c(2, 3))
```

```
[1] "long"  "long"  "short" "short" "short"
```

```
rep(c(TRUE, FALSE), c(6, 4))
```

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

3 Extracting and Assigning Vector Elements

3.1 Subsetting

Square brackets are used to extract specific parts of data from objects in R. Extracting data this way is also called **subsetting**. We input the index of the element(s) we want to extract.

To illustrate subsetting, we will consider the following example.

To keep his body in (literally perfect) shape, Chris Traeger runs 10k every day. His running times (in minutes) for the last ten days are:

$$51, 40, 57, 34, 47, 50, 50, 56, 41, 38$$

We first input the data into R as a vector and save it as the `running_times` object.

```
# Input the data into R
running_times <- c(51, 40, 57, 34, 47, 50, 50, 56, 41, 38)
# Print the values
running_times
```

```
 [1] 51 40 57 34 47 50 50 56 41 38
```

3.1.1 Positive Indices

Recall that the `[1]` in front of the output is an index, telling us that 51 is the first element of the vector `running_times`. By counting across the vector, we can see, for example, that the 5th element of `running_times` is 47. More efficiently, we can extract just the 5th element by typing `running_times[5]`.

```
running_times[5] # Extract the 5th element
```

```
[1] 47
```

To extract multiple values at once, we can input a vector of indices:

```
running_times[c(3, 7)] # Extract the 3rd and 7th elements
```

```
[1] 57 50
```

```
running_times[4:8] # Extract the 4th through 8th elements
```

```
[1] 34 47 50 50 56
```

Reordering the indices will reorder the elements in the vector:

```
running_times[8:4] # Output the 4th through 8th elements in reverse order
```

```
[1] 56 50 50 47 34
```

3.1.2 Negative Indices

Negative indices allow us to avoid certain elements, extracting all elements in the vector *except* the ones with negative indices.

```
running_times[-4] # Output all elements except the 4th one
```

```
[1] 51 40 57 47 50 50 56 41 38
```

```
running_times[-c(1, 5)] # Output all elements except the 1st and 5th
```

```
[1] 40 57 34 50 50 56 41 38
```

```
running_times[-(1:4)] # Output all elements except the first four
```

```
[1] 47 50 50 56 41 38
```

Note: Notice that `-(1:4)` is not the same as `-1:4`.

Using a zero index outputs nothing. A zero index is not commonly used, but it can be useful to know for more complicated expressions.

```
index_vector <- 0:5 # Create a vector of indices
running_times[index_vector] # Extract the values corresponding to the index.vector
```

```
[1] 51 40 57 34 47
```

Caution: Do not mix positive and negative indices.

```
running_times[c(-1, 3)]
```

```
Error in running_times[c(-1, 3)]: only 0's may be mixed with negative subscripts
```

The issue with indices of mixed signs is that R does not know the order in which the subsetting should occur: Do we want to output the third element before or after removing the first one?

Question: How could we code outputting the third element of `running_times` after removing the first one?

3.1.3 Fractional Indices

Always use integer valued indices. Fractional indices will be truncated towards 0.

```
running_times[1.9] # Outputs the 1st element (1.9 truncated to 1)
```

```
[1] 51
```

```
running_times[-1.9] # Outputs everything except the 1st element (-1.9 truncated to -1)
```

```
[1] 40 57 34 47 50 50 56 41 38
```

```
running_times[0.5] # Outputs an empty vector (0.5 truncated to 0)
```

```
numeric(0)
```

Note: The output `numeric(0)` is a numeric vector of length zero.

3.1.4 Blank Indices

Subsetting with a blank index will output everything.

```r
running_times
```

```
 [1] 51 40 57 34 47 50 50 56 41 38
```

```r
running_times[] # Same output
```

```
 [1] 51 40 57 34 47 50 50 56 41 38
```

Blank indices will be important later (when we have ordered indices).

3.2 Assigning Values to an Existing Vector

Suppose Chris Traeger made a mistake in recording his running times. On his fourth run, he ran 10k in 43 minutes, not 34 minutes. Rather than reentering all of his running times, how can we modify the existing `running_times` vector?

R allows us to assign new values to existing vectors by again using the assignment operator `<-`. Rather than specifying a new object name on the left of the assignment, we can put the element or elements in the named vector that we want to change.

```r
# Display Chris Traeger's running times
running_times
```

```
 [1] 51 40 57 34 47 50 50 56 41 38
```

```r
# Assign 43 to the 4th element of the running_times vector
running_times[4] <- 43
# Verify that the running_times vector has been updated
running_times
```

```
 [1] 51 40 57 43 47 50 50 56 41 38
```

If Chris found that the last two values were also incorrect, we can reassign multiple values at once using vector indices.

```r
# Assign 42 to the 9th element and 37 to the 10th element
running_times[9:10] <- c(42, 37)
# Verify that the running_times vector has been updated
running_times
```

```
 [1] 51 40 57 43 47 50 50 56 42 37
```

Note: The original value of 34 in the `running_times` vector has been overwritten, so reassigning values to an existing object is irreversible. Depending on the situation, it might be beneficial to first make a copy of the original data as a separate object before making changes This ensures that the original data is still retrievable if there is a mistake in the modifications.

Caution: You cannot use this syntax to create a new object. For example, the following code will not work:

```r
bad[1:2] <- c(4, 8)
```

```
Error in bad[1:2] <- c(4, 8): object 'bad' not found
```

The reason why this gives an error is that extracting or assigning individual vector elements using square brackets is actually done through functions (remember: everything is a function call). R cannot apply the extract/assign function to a vector that does not exist. The vector needs to be created first.

The following code fixes the issue:

```
good <- numeric(2) # Create an empty vector of length 2
good[1:2] <- c(4, 8)
good
```

```
[1] 4 8
```

Note: The `numeric()`, `character()`, and `logical()` functions can create empty vectors of a specified length for their respective modes. The default elements will all be `0`, `""`, and `FALSE`, respectively.

```
numeric(3) # Create a numeric vector of length 3
```

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

```
character(5) # Create a character vector of length 5
```

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

```
logical(4) # Create a logical vector of length 4
```

```
[1] FALSE FALSE FALSE FALSE
```

Creating empty or blank vectors will be important when working with for and while loops.

4 Vector Arithmetic

Arithmetic can be done on numeric vectors using the usual arithmetic operations. The operations are applied elementwise, i.e., to each individual element.

For example, if we want to convert Chris Traeger's running times from minutes into hours, we can divide all of the elements of `running_times` by 60.

```
# Divide the running times by 60
running_times_in_hours <- running_times / 60
# Print the running_times_in_hours vector
running_times_in_hours
```

```
 [1] 0.8500000 0.6666667 0.9500000 0.7166667 0.7833333 0.8333333 0.8333333
 [8] 0.9333333 0.7000000 0.6166667
```

Here are some other examples:

```
# Create a vector of the integers from 1 to 10
first_ten <- 1:10
# Subtract 5 from each element
first_ten - 5
```

```
 [1] -4 -3 -2 -1  0  1  2  3  4  5
```

```
# Square each element
first_ten^2
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

Arithmetic operations can also be applied between two vectors. Just like with scalars, the binary operators work element-by-element.

For example:

```
x <- c(1, 3, 5) # Create a sample x vector
y <- c(2, 4, 3) # Create a sample y vector
```

```r
# Add x and y
x + y
```

```
[1] 3 7 8
```

```r
# Multiply x and y
x * y
```

```
[1]  2 12 15
```

```r
# Exponentiate x by y
x^y
```

```
[1]   1  81 125
```

Symbolically, if $x = (x_1, x_2, x_3)$ and $y = (y_1, y_2, y_3)$ are vectors, then vector arithmetic in R would output:

- $x + y = (x_1 + y_1, x_2 + y_2, x_3 + y_3)$
- $x - y = (x_1 - y_1, x_2 - y_2, x_3 - y_3)$
- $x * y = (x_1 * y_1, x_2 * y_2, x_3 * y_3)$
- $x/y = (x_1/y_1, x_2/y_2, x_3/y_3)$
- $x^y = (x_1^{y_1}, x_2^{y_2}, x_3^{y_3})$

Side Note: This is *not* how vector operations work in vector calculus or linear algebra. In those fields, only addition and subtraction can be applied between vectors. Standard multiplication, division, and exponentiation do not make sense.

4.1 Recycling

When applying arithmetic operations to two vectors of different lengths, R will automatically **recycle**, or repeat, the shorter vector until it is long enough to match the longer vector.

For example:

```r
c(1, 3, 5) + c(5, 7, 0, 2, 9, 11)
```

```
[1]  6 10  5  3 12 16
```

```r
c(1, 3, 5, 1, 3, 5) + c(5, 7, 0, 2, 9, 11) # This is the same computation that R did
```

```
[1]  6 10  5  3 12 16
```

The basic arithmetic involving a vector and a scalar (i.e., a vector of length one) is implicitly using recycling.

```r
c(1, 3, 5) + 5
```

```
[1]  6  8 10
```

```r
c(1, 3, 5) + c(5, 5, 5) # This is the computation that R did
```

```
[1]  6  8 10
```

Caution: When the length of the longer vector is a multiple of the length of the smaller one, R does not give any indication that it needed to recycle the shorter vector. It is up to the user to know how the operation is interpreted by R.

If the length of the longer vector is not a multiple of the length of the smaller one, the operation will still be executed, but R will also output a warning. The warning is meant to alert the user in case the mismatched vector lengths are due to a mistake in the code.

```
c(1, 3, 5) + c(5, 7, 0, 2, 9)
```

```
Warning in c(1, 3, 5) + c(5, 7, 0, 2, 9): longer object length is not a multiple
of shorter object length
```

```
[1]  6 10  5  3 12
```

```
c(1, 3, 5, 1, 3) + c(5, 7, 0, 2, 9) # This is the computation that R did
```

```
[1]  6 10  5  3 12
```

Note: Notice the difference between warnings and errors. When a warning is given, R still executes the preceding command. When an error is given, R does not execute the preceding command.

Side Note: Recycling is not done in vector calculus or linear algebra. Vectors are required to have the same length (i.e., be of the same dimension) to be added or subtracted.

5 Vectorization

Suppose we have a function that we want to apply to all elements of a vector. In many cases, functions in R are **vectorized**, which means that applying a function to a vector will automatically apply the function to each individual element in the vector.

Vector arithmetic actually implements vectorized operations. Thus, any function created using vectorized operations is also vectorized. For example:

```
squared_dev <- function(x, c) {
  # This function inputs a vector x and a scalar c
  # and outputs a vector of squared deviations from c.
  (x - c)^2
}
# Compute squared deviations from 0
squared_dev(x = 1:5, c = 0)
```

```
[1]  1  4  9 16 25
```

```
# Compute squared deviations from 3
squared_dev(x = 1:5, c = 3)
```

```
[1] 4 1 0 1 4
```

Notice how `squared_dev()` is a vectorized function built out of two vectorized arithmetic operations.

The built-in mathematical functions we considered in the previous chapter are also vectorized:

```
sqrt(1:30)
```

```
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
 [9] 3.000000 3.162278 3.316625 3.464102 3.605551 3.741657 3.872983 4.000000
[17] 4.123106 4.242641 4.358899 4.472136 4.582576 4.690416 4.795832 4.898979
[25] 5.000000 5.099020 5.196152 5.291503 5.385165 5.477226
```

```
log(1:30)
```

```
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851 2.3978953 2.4849066 2.5649494 2.6390573
[15] 2.7080502 2.7725887 2.8332133 2.8903718 2.9444390 2.9957323 3.0445224
[22] 3.0910425 3.1354942 3.1780538 3.2188758 3.2580965 3.2958369 3.3322045
[29] 3.3672958 3.4011974
```

```r
exp(1:30)
```

```
 [1] 2.718282e+00 7.389056e+00 2.008554e+01 5.459815e+01 1.484132e+02
 [6] 4.034288e+02 1.096633e+03 2.980958e+03 8.103084e+03 2.202647e+04
[11] 5.987414e+04 1.627548e+05 4.424134e+05 1.202604e+06 3.269017e+06
[16] 8.886111e+06 2.415495e+07 6.565997e+07 1.784823e+08 4.851652e+08
[21] 1.318816e+09 3.584913e+09 9.744803e+09 2.648912e+10 7.200490e+10
[26] 1.957296e+11 5.320482e+11 1.446257e+12 3.931334e+12 1.068647e+13
```

Note: The `e+` notation in the output of `exp(1:30)` represents scientific notation. The value for `exp(30)` means 1.068647×10^{13}.

Clever use of vectorized operations and functions can make computations in R more efficient than using a loop (discussed in a later chapter) to apply functions individually to each element.

5.1 The `vapply()` Function

While many functions in R are vectorized by definition, there are also non-vectorized functions that we may want to apply to each element of a vector. One efficient way to vectorize a non-vectorized function is with the `vapply()` function.

The **`vapply()`** function applies a function to each element of a vector (or a list, which will be discussed later). The syntax of `vapply()` is `vapply(X, FUN, FUN.VALUE, ...)`, where the arguments are:

- `X`: A vector or a list.
- `FUN`: The function to be applied.
- `FUN.VALUE`: A "template" vector that specifies the type of output value you expect the `FUN` function to output.
- `...`: Any optional arguments to be passed to the `FUN` function (for example, `na.rm = TRUE`).

If the applied function in the `FUN` argument of `vapply()` outputs a single value, the output of the `vapply()` function will be a vector. If the applied function outputs a vector (with more than one element), then the output of `vapply()` will be a matrix (a two-dimensional array of values). We will learn more about matrices in a later chapter.

5.1.1 Example: Vectorizing a Non-Vectorized Function

The **`isTRUE()`** function is used to determine if the input object is identically equal to the logical value `TRUE`.

```r
isTRUE(TRUE)
```

```
[1] TRUE
```

```r
isTRUE(FALSE)
```

```
[1] FALSE
```

```r
isTRUE(NA)
```

```
[1] FALSE
```

The `isTRUE()` function checks if the entire input object is `TRUE`, not the individual elements of the object, so the `isTRUE()` function is *not* vectorized.

```r
isTRUE(c(TRUE, FALSE, NA))
```

```
[1] FALSE
```

The output is `FALSE` because the vector `c(TRUE, FALSE, NA)` is not the single value `TRUE`.

In order to check whether each element of the vector is `TRUE`, we can vectorize the `isTRUE()` function with `vapply()`. Since the `isTRUE()` function outputs a single logical value, we would set `FUN.VALUE = logical(1)`.

```
vapply(c(TRUE, FALSE, NA), isTRUE, logical(1))
```

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

If the `FUN.VALUE` is set to a output type that is not what we are expecting, then `vapply()` will throw an error.

```
vapply(c(TRUE, FALSE, NA), isTRUE, logical(3))
```

```
Error in vapply(c(TRUE, FALSE, NA), isTRUE, logical(3)): values must be length 3,
 but FUN(X[[1]]) result is length 1
```

6 Basic Numeric Summary Functions

6.1 Built-In Functions

In statistics, one of the first steps in analyzing a numeric variable is to summarize the numeric data with descriptive statistics, such as a measure of center and a measure of spread.

There are many built-in functions to compute numeric summaries (summary statistics) for numeric vectors. Some of the most common ones are given below.

- **`sum(x)`** computes the sum of the values of `x`
- **`prod(x)`** computes the product of the values of `x`
- **`mean(x)`** computes the mean of `x`
- **`sd(x)`** computes the standard deviation of `x`
- **`var(x)`** computes the variance of `x`
- **`median(x)`** computes the median of `x`
- **`IQR(x)`** computes the interquartile range of `x`
- **`min(x)`** computes the minimum value of `x`
- **`max(x)`** computes the maximum value of `x`
- **`range(x)`** computes the minimum and maximum values of `x`
- **`diff(x)`** computes consecutive differences of `x`
- **`cumsum(x)`** computes the cumulative sum of `x`
- **`cumprod(x)`** computes the cumulative product of `x`
- **`sort(x)`** orders the values of `x` (increasing order by default)
- **`fivenum(x)`** computes the five-number summary of `x`
- **`summary(x)`** computes a few summary statistics of `x`

For example:

```
# Compute the mean of the running times
mean(running_times)
```

```
[1] 47.3
```

```
# Compute the standard deviation of the running times
sd(running_times)
```

```
[1] 6.700746
```

6.2 Example: Coding a Variance Function

As an exercise, we can verify the `var()` function by coding a variance function ourselves.

The standard formula for the sample variance of a sample of values $x_1, x_2, \ldots, x_n$ is given by

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2,$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the sample mean.

A step-by-step function to compute variance is shown below. Notice the use of vectorization within the body of the function.

```
variance <- function(x) {
  # This function inputs a vector x and outputs the variance of x.

  # Compute the sample size
  n <- length(x)

  # First compute the mean of x.
  xbar <- mean(x) # Or sum(x)/length(x)

  # Compute each deviation from the mean of x.
  devs <- x - xbar

  # Compute the squared deviations from the mean.
  squared_devs <- devs^2

  # Sum the squared deviations together.
  sum_squared_devs <- sum(squared_devs)

  # Divide and output the sum of squared deviations by n-1.
  sum_squared_devs / (n - 1)
}
```

Note: Notice that we assigned the sample size (`length(x)`) and mean (`mean(x)`) to their own objects inside the `variance()` function. If a computed value needs to be used more than once, it is more computationally efficient to compute them once and assign them to an object name. This saves the computer from having to recompute these values multiple times throughout the function or program.

The `variance()` function codes every step of the variance formula explicitly for illustrative purposes. Once you are more comfortable with vectorization and how to translate between a mathematical formula and R code, the function can be written more compactly with less object assignments. For example, the entire body of the variance function can be written in one line:

```
variance2 <- function(x) {
  # This function inputs a vector x and outputs the variance of x.
  sum((x - mean(x))^2) / (length(x) - 1)
}
```

The added space inside the `sum()` function is not necessary, but it is included here for clarity.

We can verify that these functions both give us the same answer as the built-in `var()` function.

```
var(running_times)
```

```
[1] 44.9
```

```
variance(running_times)
```

```
[1] 44.9
```

```
variance2(running_times)
```

```
[1] 44.9
```

7 Technical Subtleties

7.1 Special Values

There are a few special values to be aware of in R.

7.1.1 NA

One of the most common special values in R is the **NA** object. `NA` is used to represent missing or unknown values (NA stands for "not available"). The `NA` value has a logical mode by default, but it can be coerced into any other mode as needed.

For example, suppose Chris Traeger got the flu and missed a day of running between his fifth and sixth recorded runs. To keep the ordering in his running times, we could insert a missing value into the `running_times` vector.

```
running_times <- c(running_times[1:5], NA, running_times[6:10])
running_times
```

```
 [1] 51 40 57 43 47 NA 50 50 56 42 37
```

Missing values are important to identify and deal with for many statistical reasons. Some functions will not compute the correct value when `NA` values are present. For example:

```
mean(running_times)
```

```
[1] NA
```

```
sd(running_times)
```

```
[1] NA
```

One way to handle missing values for many built-in functions is to include the argument **`na.rm = TRUE`**, which removes `NA` values from the computations.

```
mean(running_times, na.rm = TRUE)
```

```
[1] 47.3
```

```
sd(running_times, na.rm = TRUE)
```

```
[1] 6.700746
```

7.1.2 NULL

The **NULL** object (in all caps) is used to represent an empty, undefined, or nonexistent value. Unlike the other special values, the NULL object is *not* a vector object; it has its own special mode called NULL.

```
nada <- NULL
mode(nada)
```

```
[1] "NULL"
```

```
length(nada)
```

```
[1] 0
```

Note: The use of NULL is distinct from the use of NA. NULL represents that the value does not exist, whereas NA represents a value that is existent but unknown or missing.

7.1.3 NaN

The **NaN** object is a numeric value used to represent an indeterminate form (NaN stands for "not a number"). Some functions will give a warning when an NaN is outputted, as this typically occurs when a mathematically illegal operation has been attempted. For example:

```
0 / 0
```

```
[1] NaN
```

```
log(-1)
```

```
Warning in log(-1): NaNs produced
```

```
[1] NaN
```

7.1.4 Inf

The **Inf** object is a numeric value used to represent infinity (∞). This value is outputted when a mathematical expression is actually infinite (more precisely, has an infinite limit) or is a number too large for R to store (somewhere around 10^{310}).

```
1 / 0 # Infinity
```

```
[1] Inf
```

```
log(0) # Negative infinity
```

```
[1] -Inf
```

```
exp(1000) # A non-infinite but very large number
```

```
[1] Inf
```

7.2 Approximate Storage of Numbers

7.2.1 Floating Point Representation

Computers are unable to represent all real numbers with infinite precision. For example, a computer is unable to store the true value of the irrational number $\pi \approx 3.1415927$. While a computer is technically able to represent rational numbers exactly, it is more common to use an approximate representation.

Humans represent numbers and perform arithmetic calculations using the decimal number system. In decimal representation, a positive number a is expressed as

$$r = \sum_k a_k 10^k,$$

where $a_k \in \{0, 1, 2, \ldots, 9\}$ are the digits of a, and 10 is the base of the number system.

For example, the number 5413.29 can be expressed as

$$5413.29 = 5 \times 10^3 + 4 \times 10^2 + 1 \times 10^1 + 3 \times 10^0 + 2 \times 10^{-1} + 9 \times 10^{-2}.$$

R, and most other computer programming languages, use **floating point representation**, which is a binary (base 2) variation on scientific notation.

For example, consider a number written to four significant digits as 6.926×10^{-4}. This approximate representation could represent any true value between 0.00069255 and 0.00069265. In floating point representation, the significant digits are written in binary notation, and the power of 10 is replaced by a power of 2.

In the binary number system, digits are either 0 or 1. So 6.926×10^{-4} is written as $1.011_2 \times 2^{-11}$. The subscript of 2 in 1.011_2 denotes base 2. The number 1.011_2 represents

$$1.011_2 = 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 1.375.$$

Even though 6.926×10^{-4} is written as $1.011_2 \times 2^{-11}$, they are not identical representations. It turns out that four binary digits have less precision than four decimal digits. The representation of $1.011_2 \times 2^{-11}$ could represent any true value between about about 0.000641 and 0.000702. The number 6.926×10^{-4} actually does not have an exact binary representation in a finite number of digits.

The standard precision in R, known as **double precision** in computer science, is 53 binary digits (or bits), which is equivalent to about 15 or 16 decimal digits. Floating point numbers using double precision are sometimes called **doubles**. Whole numbers, or **integers**, are often stored using 32 bit integer storage.

Note: In some programming languages, integers and doubles are considered separate numeric types. R actually also has separate integer and double types, but R will automatically switch between them to make computations easier. The "numeric" mode in R is a synonym for the double type (both names exist in R as a historical artifact). The "integer" type is technically separate from numeric, but calling `mode()` on an integer vector will still output `"numeric"`.

Side Note: The **`typeof()`** function outputs the **internal storage type** of an input object. This is the same as the mode of an object, except that the integer and double types both have numeric modes.

```
typeof(first_ten)
```

```
[1] "integer"
```

```
mode(first_ten)
```

```
[1] "numeric"
```

```
typeof(pi)
```

```
[1] "double"
```

```r
mode(pi)
```

```
[1] "numeric"
```

7.2.2 Rounding Errors

The reason why we need to understand that numbers are stored approximately in R is that this inherent limitation of finite precision representations affects the accuracy of calculations. Any computations done on approximated numbers can accumulate **rounding errors**.

For example, using exact arithmetic, we know that $(5/4) \times (4/5) = 1$, so $(5/4) \times (n \times 4/5) = n$, for any number n. However, this simple calculation in R already has rounding errors:

```r
n <- 1:10
1.25 * (n * 0.8) - n
```

```
 [1] 0.000000e+00 0.000000e+00 4.440892e-16 0.000000e+00 0.000000e+00
 [6] 8.881784e-16 8.881784e-16 0.000000e+00 0.000000e+00 0.000000e+00
```

The exact answer should be 0 for all n, but we see that there are errors for some values of n. The errors in this example are essentially negligible (around 10^{-16}), but they are important to be aware of and acknowledge. Rounding errors tend to occur in most computations, so long series of computations tend to accumulate larger errors than shorter ones.

7.2.2.1 The Variance Function Revisited

A common statistical example to illustrate the effect of rounding errors is in computing the variance.

The standard formula for the variance requires calculating the sample mean $\bar{x}$ first and then the sum of squared deviations second, so the computer needs to cycle (or *pass*) over the data values twice. This is considered computationally expensive, since it requires storing the data values in the computer's memory between passes over the data.

Through some algebraic manipulation, an alternate "one-pass'' formula is given by

$$s^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right).$$

While this formula is mathematically equivalent and computationally less expensive, it can be numerically unstable when the variance is small relative to the mean. To illustrate this, we will first write the one-pass function.

```r
var_one <- function(x) {
  # This function inputs a vector x
  # and computes the one-pass formula for the variance of x.
  n <- length(x)
  (sum(x^2) - n * mean(x)^2) / (n - 1)
}
```

This function will give the correct answer for small examples.

```r
var(first_ten) # Built-in function
```

```
[1] 9.166667
```

```r
variance(first_ten) # Two-pass function
```

```
[1] 9.166667
```

```r
var_one(first_ten) # One-pass function
```

```
[1] 9.166667
```

Suppose we add a large value (like 10^{10}) to every value in `first_ten`.

Question: What happens to the mean when we add a large value to every value? What about to the variance?

```r
# Add 10^10 to the first_ten vector and assign the result to the uhoh vector
uhoh <- first_ten + 1e10
var(uhoh) # Built-in function
```

```
[1] 9.166667
```

```r
variance(uhoh) # Two-pass function
```

```
[1] 9.166667
```

```r
var_one(uhoh) # One-pass function
```

```
[1] 0
```

Since the inner terms $\sum_{i=1}^{n} x_i^2$ and $n\bar{x}^2$ are very close when the x_i values are large, then computing the difference results in a drastic loss of precision.

Caution: Do not use the one-pass variance formula in practice.