# Coding a Sum Function

Appendix to Chapter 7

Stats 20: Introduction to Statistical Programming with R

UCLA



## Contents

**Learning Objectives** **1**

**1 Example: Coding a Sum Function** **1**

1.1 How Does the Built-in Function Work? . . . . . 2
1.2 One Vector Input . . . . . 2
1.3 Using the ... Argument . . . . . 3
1.4 Input Validation and Error Handling . . . . . 4
1.5 Examples . . . . . 5





Acknowledgements: Jake Elmstedt



## Learning Objectives



After studying this section, you should be able to:

- Write a function that accommodates a variable number of arguments with the `...` argument.

## 1 Example: Coding a Sum Function

As an application of the material covered in Chapter 7 (specifically data frames, lists, and the `apply` family of functions), we consider the example of writing our own sum function that mimics the functionality of the built-in `sum()` function.

When writing a function that already exists, we want to think through the problem by asking ourselves a series of questions:

- How does the built-in function work?
- How does it work for one vector input?
- How do we account for NAs?
- How do we generalize to accommodate multiple arguments?
- How do we allow for data frames (and warn of coercion)?
- How do we throw errors if the inputs are not summable?

## 1.1 How Does the Built-in Function Work?

To familiarize ourselves with the functionality we want out of our own function, we want to see how the built-in function works. The two main ways to get a better understanding of a built-in function are to look up the R documentation and to try different inputs, including edge cases.

From the help file (`?sum`), we first see the description of what the `sum()` function does:

> `sum` returns the sum of all the values present in its arguments.

The syntax for the `sum()` function is `sum(..., na.rm = FALSE)`, where the arguments are

- `...`: numeric or complex or logical vectors.
- `na.rm`: a logical value specifying whether missing values (including `NaN`) should be removed.

In its most basic form, for a single vector input, then `sum()` will add all the values in the vector. The entire sum is `NA` if any `NA` or `NaN` values are present and `na.rm = FALSE`. If there are only `NA`s and `na.rm = TRUE`, then the sum is `0`.

```
sum(c(1, 3, 5, 7, 9))
```

```
[1] 25
```

```
sum(c(1, 3, 5, 7, 9, NA, NaN))
```

```
[1] NA
```

```
sum(c(1, 3, 5, 7, 9, NA, NaN), na.rm = TRUE)
```

```
[1] 25
```

```
sum(NA, na.rm = TRUE)
```

```
[1] 0
```

```
sum(numeric(0))
```

```
[1] 0
```



Notice, however, that the `sum()` function is not restricted to take in only single vector inputs. That is, the main argument is not a single vector `x` but rather `...`, which allows for any number of (unnamed) arguments to be summed. When more than one argument is inputted, the `sum()` function will sum *all* of the values.

```
sum(c(1, 3, 5), 7, 9)
```

```
[1] 25
```

In this example, the `sum()` function takes in three separate vector arguments rather than a single vector.

To write a function that achieves this flexible functionality, we first want to write a version that works for one vector input and then extend the function to work for multiple inputs.

## 1.2 One Vector Input

As a stepping stone to our final `sum()` function, we will write a `sum_helper()` function that allows for a single vector input.

If all the values to sum are in one vector argument `x`, we simply need to iteratively add all of the values together. A basic description of the logic is as follows:

- If `na.rm = TRUE`, then we want to remove any `NA` or `NaN` values before summing.
- Start from a sum of 0.
- For each value in `x`, add the value to the sum.

- Return the sum.

Translating this description directly into R code, the `sum_helper()` can be written as:

```
sum_helper <- function(x, na.rm = FALSE) {
  if (na.rm) {
    x <- x[!is.na(x)]
  }
  out_sum <- 0
  for (i in x) {
    out_sum <- out_sum + i
  }
  out_sum
}
```



Notice that `sum_helper()` already handles the edge cases of length 0 vectors and vectors with `NA` or `NaN` values.

### 1.3 Using the `...` Argument



Now that we have a working function for single vector inputs, we want to write a function that accommodates multiple inputs.



The `...` argument is used to allow for any number of arguments. That is, by writing `function(...)`, a function can take in multiple arguments rather than being constrained to a fixed number with predefined named arguments like `function(x)` or `function(x, y, z)`.



The difficulty in allowing multiple arguments is that we do not know a priori how many arguments the user will input, and all the arguments may have different lengths or types. We need a way of storing all of the separate arguments into a single object within the body of our function.

The main trick is to encapsulate the inputs of `...` into a list, since a list object will store any number of objects and preserve the types of each one. The first command in the body of our function will be to store a local object `list(...)` that contains every argument that is inputted into our function.



**Note**: The `na.rm` argument is a named argument that is separate from the `...` argument. Since the number of arguments inputted into our function will be variable, we cannot consider `na.rm` as the "second" argument. Thus the `na.rm` argument must be specified by name rather than position.

A basic description of the logic for our extended sum function is as follows:

- Store all the inputs into a list called `args`.
- Compute the sum for each component in `args` separately using the `sum_helper()` function.
- Sum all the component sums together.
- Return the grand sum of all the components.

Translating this description into R code, the `my_sum()` can be written as:

```
my_sum <- function(..., na.rm = FALSE) {
  args <- list(...)
  args_sum <- vapply(args, sum_helper, numeric(1), na.rm = na.rm)
  sum_helper(args_sum)
}
```

Note that we used `vapply()` on the `args` list rather than a loop, since we are applying the `sum_helper()` function to each component separately. In addition, the `na.rm` argument of `sum_helper()` inherits the input from the `na.rm` argument specified in the `my_sum()` function call.

### 1.4 Input Validation and Error Handling

At this point, we have a working `my_sum()` function for vectors and matrices (since matrices will be interpreted by R as a vector). To complete our function, we want to be able to validate our inputs and throw errors or warnings when the arguments are not what we expect. In general, any warnings or errors that is thrown from a function you wrote should be written by you. Your function throwing a built-in error means that you did not account for a particular edge case.

When considering input validation, we want to think about what types of objects are valid inputs and what types are not. Certainly logical or numeric vectors and matrices would be considered valid inputs (complex inputs are fine as well, but we will not account for these explicitly), but character vectors would not be valid. We will also not allow list inputs. Invalid inputs into our function should throw an error.

To check whether the inputs are valid, we can use the `is.numeric()` and `is.logical()` predicate functions. Notice that these will return `TRUE` if the input has a numeric or logical storage type, which includes both vectors and matrices. We thus create an `is_summable()` helper function that checks a single input object for whether it is considered a valid summable object.



```
is_summable <- function(x) {
  is.numeric(x) || is.logical(x) # || is.complex(x)
}
```



Data frames that store entirely numeric or logical values should be considered a valid input, but data frames are not interpreted by R in the same way as matrices, so our code would not work as intended. Therefore, it is reasonable to first coerce data frames into matrices before decomposing the sum. It is good practice to throw a warning when coercing an input to warn the user that the function is making an assumption about how the user may want the function to interpret its input.



We thus will include checking whether any input argument is a data frame by using the `is.data.frame()` predicate function, which returns `TRUE` if the object is a data frame and `FALSE` otherwise. If any argument is a data frame, we will coerce all of them to a matrix type with the `as.matrix()` type casting function.



Putting everything together, a fully formed working `my_sum()` function including input validation and error handling is given below:



```
my_sum <- function(..., na.rm = FALSE) {
  args <- list(...)
  if (any(vapply(args, is.data.frame, logical(1)))) {
    args <- lapply(args, as.matrix)
    warning("Coerced data frames to matrix type")
  }
  if (!all(vapply(args, is_summable, logical(1)))) {
    stop("Invalid argument type(s)")
  }
  args_sum <- vapply(args, sum_helper, numeric(1), na.rm = na.rm)
  sum_helper(args_sum)
}
```

**Question**: Why do we check for and coerce data frames before checking for summable inputs? Can the order be switched?

### 1.5 Examples

Below are examples that demonstrate the functionality of our `sum_helper()` and `my_sum()` functions on different inputs to showcase the intended results.

```
sum_helper(trees)
```

```
 [1]  88.6  83.9  82.0  98.9 110.5 113.5  92.6 104.2 113.7 106.1 114.5 108.4
[13] 108.8 102.0 106.1 109.1 131.7 126.7 110.4 102.7 126.5 125.9 124.8 126.3
[25] 135.9 153.7 155.2 156.2 149.5 149.0 184.6
```

```
sum_helper(numeric())
```

```
[1] 0
```

```
sum_helper(NA, na.rm = TRUE)
```

```
[1] 0
```

```
sum_helper(1:100)
```

```
[1] 5050
```

```
sum_helper(c(1:10, NA))
```

```
[1] NA
```

```
sum_helper(c(1:10, NA), na.rm = TRUE)
```

```
[1] 55
```

```
my_sum(trees)
```

```
Warning in my_sum(trees): Coerced data frames to matrix type
```

```
[1] 3702
```

```
my_sum(trees, 41)
```

```
Warning in my_sum(trees, 41): Coerced data frames to matrix type
```

```
[1] 3743
```

```
my_sum()
```

```
[1] 0
```

```
my_sum(c(FALSE, TRUE, TRUE), 14)
```

```
[1] 16
```

```
my_sum(NA, na.rm = TRUE)
```

```
[1] 0
```

```
my_sum(14, NA, 1:10, c(1, NA), na.rm = TRUE)
```

```
[1] 70
```

```
my_sum(list(1:10))
```

```
Error in my_sum(list(1:10)): Invalid argument type(s)
```

```
my_sum("A", "B", "C")
```

```
Error in my_sum("A", "B", "C"): Invalid argument type(s)
```