# Introduction to Functions and {purrr}

2025-06-09

## When to write a function

R comes with thousands of functions, and more are written everyday (these are published in R packages). If you want to do something in R, there is a good chance that a function already exists somewhere that does it.

But what if you want to do something new in R, something that doesn't yet have a function? What should you do?

Write a function!

This tutorial will show you how.

## Workflow

How should you turn your code into functions? Always follow these four steps:

1. Create a real R object (or set of objects) to use with your function
2. Write code that works with the real object(s)
3. Wrap the code in `function()`
4. Assign the names of your real objects as argument names to the function

These steps are the best practice for writing functions in R. When you follow these steps, your functions are guaranteed to work for at least one case, which means you will spend more time using your functions and less time debugging them.

Let's use the steps to create your first function.

### Goal - Grading

To make this real, put yourself in the shoes of a teacher:

You've given your students 10 homework assignments and announced that you will drop their lowest homework score. Their final grade will be the average score of the remaining homeworks.

To make your life easier, you want to write an R function that will take a vector of 10 homework scores and return a final grade.

Ready to begin?

Remember our steps:

1. Create a real R object (or set of objects) to use with your function

- Create an object named `x` that contains the vector `c(100, 100, 100, 100, 100, 100, 100, 100, 100, 90)` (hint: use copy and paste!).

```
x <- c(100, 100, 100, 100, 100, 100, 100, 100, 100, 90)
```

2. Write code that works with the real object(s)

Recall the grading scheme: the final grade is the average homework score after you drop the lowest score. Since there are many ways to calculate this, let's be specific and end up on the same page.

- Use `sum()`, `min()`, `/`, `9` and parentheses to calculate the final grade for student `x`.

```
(sum(x) - min(x)) / 9
```

```
## [1] 100
```

3. Wrap the code in `function()`

The `function()` function builds a function from a piece of R code. To use it, call `function()` followed by an opening brace, `{`. Then write one or more lines of R code followed by a closed brace, `}`, e.g.

```
foo <- function() {
 a <- 1
 b <- 2
 a + b
}
```

`function()` will return a function that uses everything between the braces as its code body. If you'd like to save the function, you'll need to assign it in the usual way to an R object that you can call later.

As you write your functions, recall that R will only return the result of the last line in the code body when you call the function (we'll learn about some exceptions to this rule in the Control Flow tutorial).

Once you save a function, you can run it and inspect its contents.

```
foo()
```

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

```
foo
```

```
## function() {
##  a <- 1
##  b <- 2
##  a + b
## }
```

### Your turn

Let's save your code as a function.

```
grade <- function() {
  (sum(x) - min(x)) / 9
}
```

At the moment, your `grade()` function is reusable but not *generalizable*. Each time you call `grade()` it computes the final grade of the vector `x` that contains `c(100, 100, 100, 100, 100, 100, 100, 100, 100, 90)`.

```
grade()
```

```
## [1] 100
```

```
grade()
```

```
## [1] 100
```

We'd like to use `grade()` with new vectors that have new values.

### Arguments

4. Assign the names of your real objects as argument names to the function

You can make a function generalizable by turning some of the objects in its code body into *formal arguments*. A formal argument is an object that a user can assign a value to when he or she calls the function. The function will use the user's value for the object when it executes its code body.

For example, we'd like to tell R that `x` in `grade()` is an argument. R shouldn't use a pre-defined value for `x`; it should let the user supply a new value for `x` each time he or she runs the function.

```
grade <- function() {
  (sum(x) - min(x)) / 9
}
```

How do you tell R that an object is a formal argument?

You list the name of the object in the parentheses that follow `function()` in the function definition. If you make more than one argument, separate their names with a comma. For example, you could make `a` and `b` arguments of my `foo` function.

```
foo <- function(a, b) {
 a + b
}
```

Now I can define a new value for `a` and `b` each time I call `foo`.

```
foo(a = 1, b = 1)
```

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

```
foo(a = 100, b = 200)
```

```
## [1] 300
```

### Default values

To give an argument a default value, set it equal to a value when you define the function. For example, the code below will set the default value of `b` in `foo` to one.

```
foo <- function(a, b = 1) {
 a + b
}
foo(a = 2)
```

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

```
foo(a = 2, b = 2)
```

```
## [1] 4
```

Interesting, huh? Now apply what you've learned to `grade()`.

- Change the code below to list `x` as a formal argument of `grade()`.

```
grade <- function(x) {
  (sum(x) - min(x)) / 9
}
```

You can now have a finished `grade()` function that you can use to calculate the final grade of *any* vector (I mean, student). Try it out.

- Calculate the final grade of the vector `c(100, 90, 90, 90, 90, 90, 90, 90, 90, 80)`.

```
grade(x = c(100, 90, 90, 90, 90, 90, 90, 90, 90, 80))
```

```
## [1] 91.11111
```

### Recap

Use the four step workflow whenever you need to write a function:

1. Create a real R object (or set of objects) to use with your function
2. Write code that works with the real object(s)
3. Wrap the code in `function()`
4. Assign the names of your real objects as argument names to the function

## {purrr} expressions

{purrr}'s map functions can take "expressions." *Expressions* are a pieces of code that are preceded by `~` and include `.x`s. Map will apply expressions iteratively to each element of a vector.

For example, this call to `map()` will apply the expression `~.x^2` to each element of `vec`. On each iteration, `map()` will assign the _i_th element of vec to `.x` and run the expression.

```
library(tidyverse)

vec <- c(1, 2, 3)
map(vec, ~.x^2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
```

In other words, {purrr}'s map expressions work like functions that have `.x` as a formal argument. To transform an expression to a function, remove the `~`, wrap the code in `function()` and list `.x` as a formal argument.

```r
sq <- function(.x) {
  .x^2
}

map(vec, sq)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
```