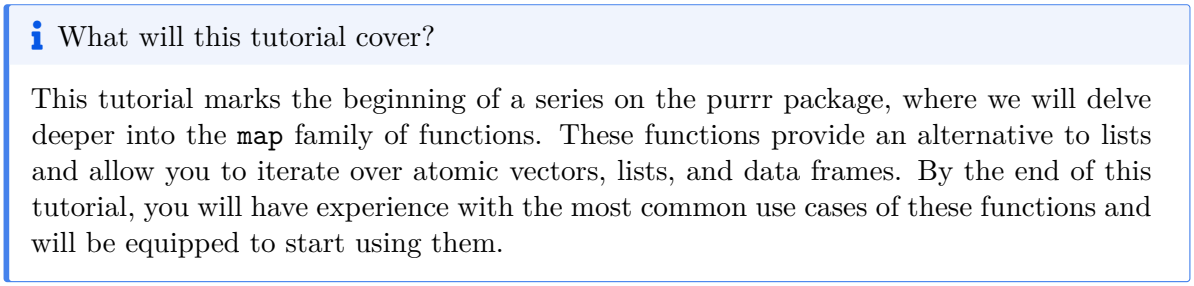
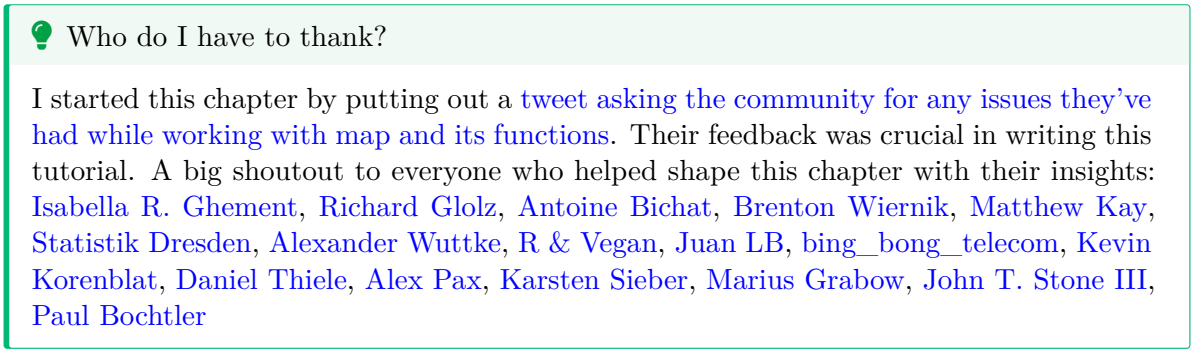
19 How to use the map function family effectively





19.1 Preparation

Before beginning this tutorial, make sure you have purrr 1.0.0 and R 4.1.0 installed. purrr 1.0.0 was released in Dezember 2022 and introduced several breaking changes to the package, while R 4.1.0, released in May, 2021, introduced the native pipe operator `|>`. This tutorial and the following will use the native pipe instead of the magrittr pipe.

19.2 A list, vector and data frame primer

Mastering purrr and the map family of functions can be challenging without a solid understanding of lists, vectors, and data frames. The `map` functions mostly take these data structures

as input. If you're not familiar with them, it's gonna be tough to follow along. I'll be talking about them a lot in this tutorial, so it's a good idea to brush up on them before diving in. Also, it can be a lot to take in all at once - data structures and the `map` functions. So, let's spend some time reviewing these data structures first.

An overview of data structures in R

R has five basic data structures: Atomic vectors, matrices, arrays, lists, and data frames (see http://adv-r.had.co.nz/Data-structures.html). They're different from each other in two ways: whether they hold the same data type or not, and whether they're one or two-dimensional. So, here's a quick rundown of these data structures:

	Homogeneous	Heterogeneous
1 dimension	Atomic vector	List
2 dimensions	Matrix	Data frame
n dimensions	Array	

Figure 19.1: R data structures

I won't be going too in-depth on matrices and arrays in this tutorial since they're not as important for purrr as the other data structures.

Atomic vectors

Atomic vectors are just one type of vector, with lists being the other. We'll talk more about lists in a bit. Both atomic vectors and lists have one thing in common: they're one-dimensional. Think of it like this: if we compare dimensions to directions, an element in a one-dimensional data structure can only move forward and backward (or up and down). But an element in a two-dimensional data structure can move in any direction. Here is a visual overview of the four types of atomic vectors (based on the book R for Data Science):

We have four types of atomic vectors: logical (`TRUE` or `FALSE`), character (text), integer (whole numbers like 1), and double (numbers with decimals like 1.31). Integers and doubles are also called numeric atomic vectors. You've all created them before. Let me show you how to create one of each type:

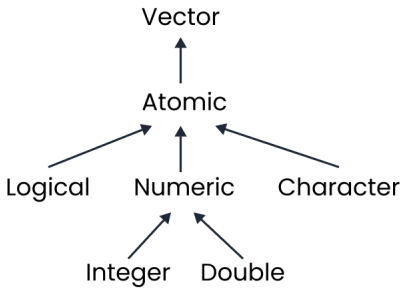


Figure 19.2: Atomic vectors

```
c(TRUE, FALSE) # logical
c("a", "bunch", "of strings") # character
c(4, 5, 9) # integer
c(5.1, 8.2) # double
```

It becomes interesting when we attempt to create non-homogeneous (or heterogeneous) vectors. By default, R will try to convert all elements of the vector to the same data type. Here is an example where we mix a double and a character:

```
c(2.1, "8")
```

```
[1] "2.1" "8"
```

The output is a character vector. In R or programming terminology, we would say the double was coerced to a character (for more information, refer to this explanation from R in a nutshell). To verify this is a character vector, we pipe it into `typeof`:

```
c(2.1, "8") |> typeof()
```

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

Coercion occurs when we combine vector elements of different types. In this example, we have combined a logical, a double, and a character:

```
c(TRUE, 8.1, "house")
```

```
[1] "TRUE"  "8.1"   "house"
```

Again, we get a character vector. What we see here are coercion rules. These rules determine in what order data types should be coerced. According to Joseph Adler in his book R in a nutshell the coercion rules in R go as follows:

> logical < integer < numeric < complex < character < list

When combining multiple data types in a vector, the data type on the left of the coercion rules will be converted to the data type on the right. For example, in the previous examples, the double and the logical was coerced to a character. This is why, when we combine a logical and a double, the logical will be converted to a double.

```
c(TRUE, 5.1, FALSE)
```

```
[1] 1.0 5.1 0.0
```

As you can see in this example, when coerced to a double, `TRUE` is represented as `1` and `FALSE` as `0`. That's all for now about atomic vectors. Next, let's discuss another one-dimensional data structure: lists.

Lists

What differentiates lists from atomic vectors are two things. First, lists are heterogeneous, which means its elements can be of different data types. Second, lists are recursive, which means that list elements can contain other objects. Here is an example of a list that shows both of these properties:

```
(my_list <- list(
  a = 3,
  b = "Some text",
  c = list(
    x = c(4, 5, 6)
  ),
  c(TRUE, FALSE)
))
```

```
$a
[1] 3
```

```
$b
[1] "Some text"

$c
$c$x
[1] 4 5 6


[[4]]
[1]  TRUE FALSE
```

In this example, the list holds three elements (`a`, `b`, and `c`). Each element is of a different data type: `a` is a numeric vector, `b` is a character vector, and `c` is a list. This is an example of a heterogeneous data structure. Recursion is also demonstrated in this example. The third element `c` contains a list, which itself contains other elements. This is not possible with atomic vectors. Recursive data structures can be extended to any depth desired:

```
list(
  c = list(
    x = list(
      z = list(
        t = c(1, 2, 3)
      )
    )
  )
)
```

```
$c
$c$x
$c$x$z
$c$x$z$t
[1] 1 2 3
```

To better read a deep data structure like this, we can use the `str()` function which provides us with information about the data types of the elements:

```
str(my_list)
```

```
List of 4
 $ a: num 3
 $ b: chr "Some text"
```

```
$ c:List of 1
 ..$ x: num [1:3] 4 5 6
$  : logi [1:2] TRUE FALSE
```

An even more readable representation of this list is a visual organizer that uses color and shape to indicate the data types of the elements:

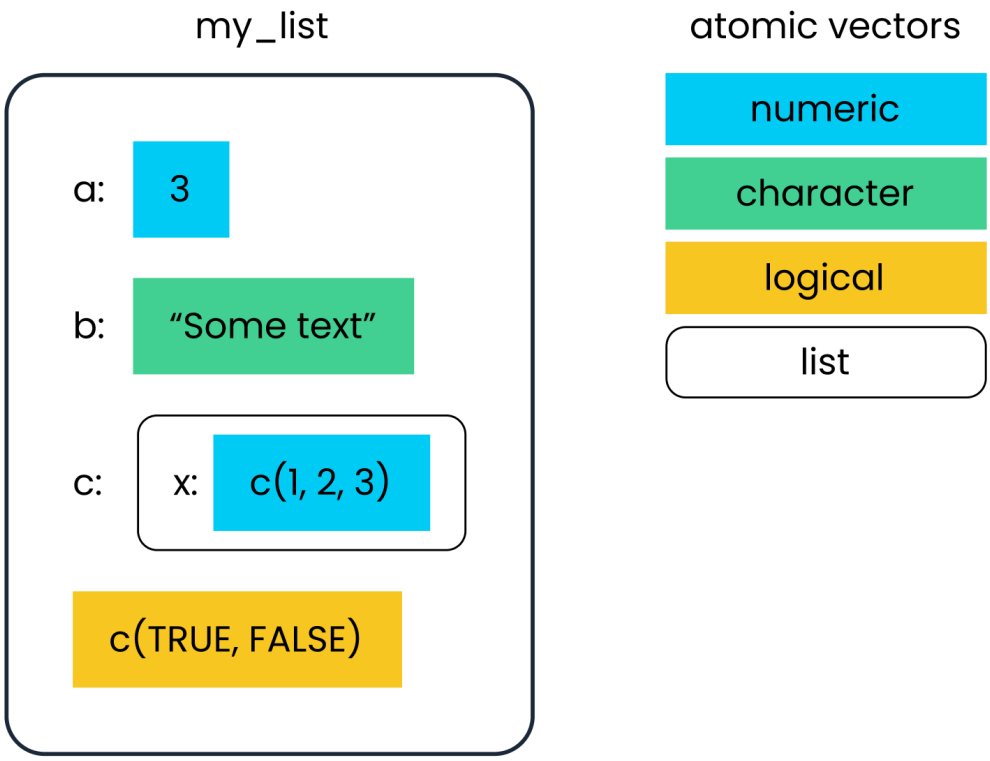


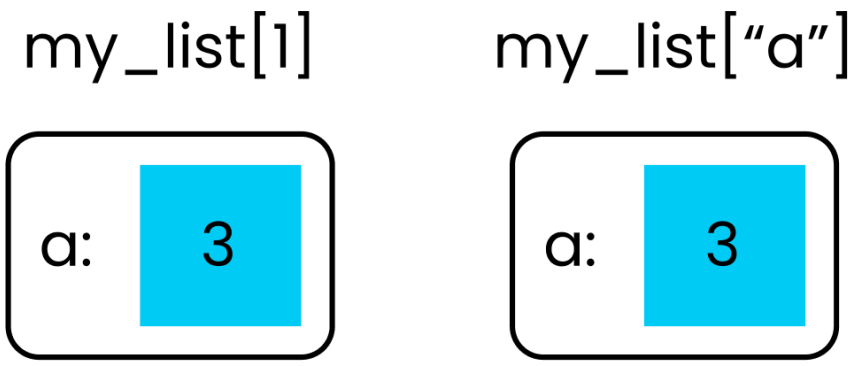
Figure 19.3: Visual depiction of a list

Direct your attention to the list elements. They are represented as white rectangles with black rounded corners. For example, the element labeled `c` is a list. Elements `a` and `x` (within `c`) contain numeric vectors and element `b` holds a character vector.

You may also notice that the last element of the list, the logical vector, does not have a name. This is valid in lists. This leads us to the next topic about lists: subsetting. Subsetting refers to the act of retrieving an element from a list. The topic may be confusing for those who are new to working with lists, so we will look at five ways of doing it first and then discuss them in more detail.

```
my_list[1]
my_list[[1]]
my_list$a
my_list[["a"]]
my_list["a"]
```

Here is one rule to remember. Whenever you subset a list element using one square bracket `[`, the result will be a list. For example, if I were to subset the first list element `a` using `my_list[1]` or `my_list["a"]`, the output will be a list. Coming back to our visual organizer, the returned list contains a numeric vector:



We can confirm this by checking the object type:

```
my_list[1] |> typeof()
```

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

```
my_list["a"] |> typeof()
```

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

Here's another rule to remember: When we subset a list element using double square brackets `[[`, we retrieve the element itself, in our case a double vector:

```
my_list[[1]] |> typeof()
```

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

```
my_list[["a"]] |> typeof()
```

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

Similarly, if a list item has a name, we can directly subset it by using its name, which is equivalent to using `[[`:

```
my_list$a
```

```
[1] 3
```

What I didn't tell you yet is how indexing works in R for lists and vectors. Indexing is a method of arranging the elements in a list or vector and assigning them a numerical value. For example:

```
my_list[[4]]
```

```
[1]  TRUE FALSE
```

What we are saying here is this: "Give me the fourth element of that list". In R, lists always start indexing at 1, so `my_list[[1]]` would return the first element of the list. Indexing is crucial for accessing elements that do not have a name. In R, attempting to index an element that is outside the range of a list will result in an error. For instance, in the given example, the list does not have a tenth element.

```
my_list[[10]]
```

```
Error in my_list[[10]] : subscript out of bounds
```

The developers of purrr have addressed this issue by creating the `pluck` function, which functions similarly to using double square brackets `[[` for lists. A useful feature of this function is that it always returns a value, even if the element doesn't exist. For example, if `my_list` only contains four elements, attempting to pick the tenth element would not result in an error. Instead `pluck` returns `NULL` for the element that doesn't exist. Here is an example:

```
pluck(
  .x = my_list,
  10
)
```

```
NULL
```

The first argument `.x` takes a list. All subsequent arguments take indices, which can be either integers or the name of an element. In the following example, I extract the first element, labeled as `x`, of the list `c` within `my_list` (have another look at `my_list` if this is not clear):

```
pluck(my_list, "c", 1)
```

```
[1] 4 5 6
```

This function call is similar to:

```
my_list[["c"]][[1]]
```

```
[1] 4 5 6
```

Data frames

Data frames are a specialized type of list that can hold heterogeneous elements. However, there are two key differences between lists and data frames. Firstly, the elements or columns of a data frame must have equal length. This is not a requirement for list elements. Secondly, data frames are two-dimensional, consisting of rows and columns. They are essentially lists of same-length vectors, as described the book Advanced R. Like lists, data frames can be subsetted by referencing the names of its elements.

```
(my_data_frame <- data.frame(
  a = c(1, 3, 4),
  b = c("A", "few", "words")
```

```
  ))
```

```
  a     b
1 1     A
2 3   few
3 4 words
```

```
  my_data_frame$a
```

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

Even the subsetting rules for data frames are similar to those for lists. Using a single set of square brackets will return a list, while double square brackets will return the specific element or column:

```
  my_data_frame[1] |> typeof()
```

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

Data frames are composed of columns that are atomic vectors, meaning they consist of a single data type. If one column of a data frame is a list, R will throw an error:

```
  data.frame(
    a = c(1, 3, 4),
    b = c("A", "few", "words"),
    c = list(a = 3, b = TRUE, c = c(3, 7))
  )
```

```
Error in data.frame(a = c(1, 3, 4), b = c("A", "few", "words"), c = list(a = 3, :
arguments imply differing number of rows: 3, 2
```

R will throw an error because the elements in the list are not of the same length as the other columns in the data frame. Even if the column is modified so that the elements have the same length, the output may not be as expected:

```
  data.frame(
    a = c(1, 3, 4),
```

```
  b = c("A", "few", "words"),
  c = list(a = 3, b = TRUE, c = 7)
)
```

```
  a     b c.a  c.b c.c
1 1     A   3 TRUE   7
2 3   few   3 TRUE   7
3 4 words   3 TRUE   7
```

Interestingly, the column `c` was spread across three columns `c.a`, `c.b`, and `c.c`. The code worked, but the list was flattened into these three columns. Despite being widely used in R for over 20 years, data frames lack some properties that make data analysis easier. To address this, the creators of the Tidverse package introduced tibbles. You can thing of tibbles as better looking and better behaving data frames.

If the same data frame was created as a tibble, it would be called a **nested tibble**. These nested tibbles will be important later in the tutorial when we will introduce the map functions. The next example shows a nested tibble, created by replacing `data.frame` with `tibble` in the previous code.

```
(my_tibble <- tibble(
  a = c(1, 3, 4),
  b = c("A", "few", "words"),
  c = list(a = 3, b = TRUE, c = c(3, 7))
))
```

```
# A tibble: 3 x 3
      a b     c
  <dbl> <chr> <named list>
1     1 A     <dbl [1]>
2     3 few   <lgl [1]>
3     4 words <dbl [2]>
```

The result is a column `c` that holds a list. This column behaves similarly to a list, with the same subsetting and indexing methods we have covered previously.

```
# Returns list element as list of length 1
my_tibble$c[1]
my_tibble$c["a"]

# Returns list element
```

```
my_tibble$c[[1]]
my_tibble$c$a
pluck(my_tibble$c, 1)
```

With this, you should now have the necessary vocabulary and concepts to get started with the map family of functions. It is important to ensure that you have fully grasped the concepts discussed up to this point, as the rest of the tutorial will build upon these foundations.

19.3 For loops in R

At the start of the tutorial, I mentioned that the map family of functions iterate over elements of certain data types. For most learners, the first technique for iterating over elements is the for loop. This is because for loops are explicit, showing each computational step clearly, and are a fundamental concept that all programmers should understand. To provide a clear connection between for loops and the map functions, I will briefly introduce for loops at this point in the tutorial.

As an example, suppose you want to use a for loop to calculate the square of each element in a numeric vector:

```
.input <- c(2, 4, 6)
output <- vector(mode = "numeric", length = length(length(.input)))

for (i in seq_along(.input)) {
  output[[i]] <- .input[[i]]^2
}

output
```

```
[1]  4 16 36
```

`.input` is an atomic vector of type numeric or integer. `output` is an empty atomic vector of type numeric with the same length as `.input`. The `length` parameter is used to allocate a specific amount of space for that vector. Initially, `output` is empty. The for loop uses `seq_along()`, which creates a sequence of values from 1 to the number of elements in `.input`. For example, if input has three elements, the sequence would be 1, 2, 3.

```
seq_along(.input)
```

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

Within the for loop, `i` corresponds to the current element of the sequence created by `seq_along()`.

```
for (i in seq_along(.input)) {
  print(i)
}
```

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

Within the body of the for loop, we calculate the square of each vector element and store the result at the same index in the output vector (`output[[i]] <- .input[[i]]^2`).

For loops and the `map` functions have some similarities and can be easily translated between, as demonstrated in the loopurrr package created by Tim Tiefenbach. They share the following characteristics:

- Both for loops and map functions take input in the form of a list, vector, data frame, or tibble.
- They both iterate over each element of these data structures and apply a computation to that element.
- They both create or return a new object.

It's important to note that these similarities only apply to the specific for loop we just created. Now that we have covered for loops, we are ready to start working with map functions.

19.4 Introduction of the map family of functions

`map` is a family of functions, consisting of at least five different functions. Similar to for loops, the map functions iterate over elements, apply a function to each element, and return the results in a data structure of your choice. The following is a visual overview of the `map` functions:

Two arguments are essential for the map functions:

- `.x`: This is either a list or an atomic vector. Data frames can also be used as input, in which case `map` will iterate over each column of the data frame.
- `.f`: A function provided to `.f` will be applied to each element of `.x`. It is considered good practice to write this function in the form of an anonymous R function (`\(x) <FUNCTION BODY>`) instead of using the tilde `~` which is considered bad practice.

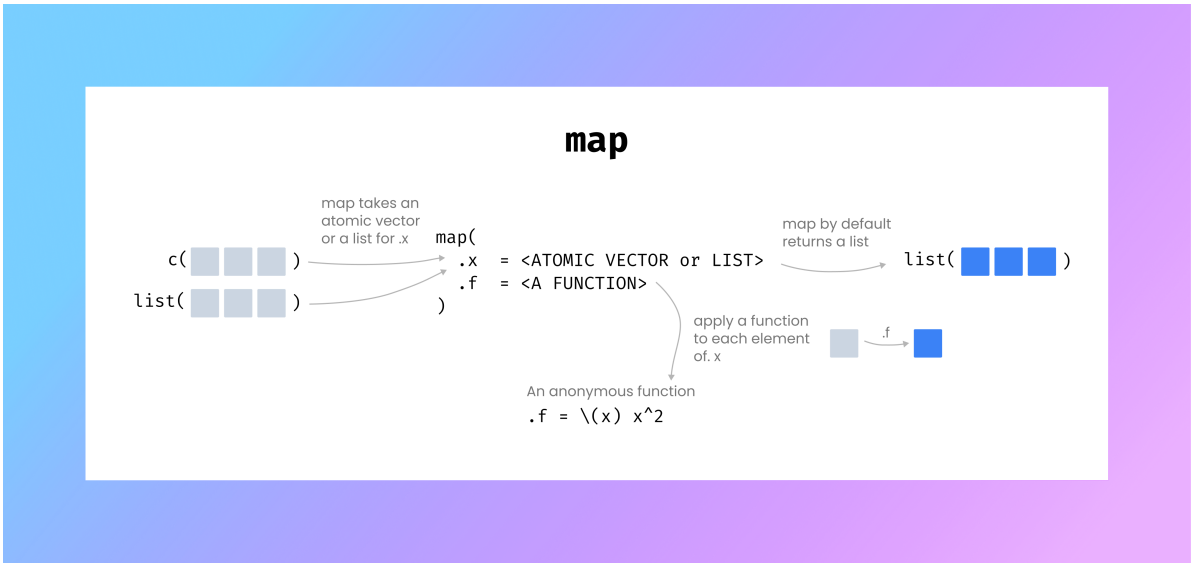


Figure 19.4: map function overview

It's important to note that `map` will always return an object that is the same length as the input. With this general understanding, let's see an example of `map`. Similar to the for loop previously, we will square each element in a numeric vector.

```
(first_map_output <- c(1, 2, 3) |> map(.f = \(x) x^2))
```

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

[[2]]
[1] 4

[[3]]
[1] 9
```

In this example, the `.x` argument is implicitly passed to map, which is why it is not visible in the code. The output has the same length as the input (3 items) and it is returned as a list:

```
first_map_output |> typeof()
```

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

What is great about the `map` functions is that the returned object does not have to be a list. The family includes different functions that specify the type of data type for the returned object by using suffixes. Here are the functions:

- `map_chr`: Returns a character vector
- `map_int`: Returns an integer vector
- `map_dbl`: Returns a double vector
- `map_lgl`: Returns a logical vector

These are the same data types that were initially introduced as atomic vectors.

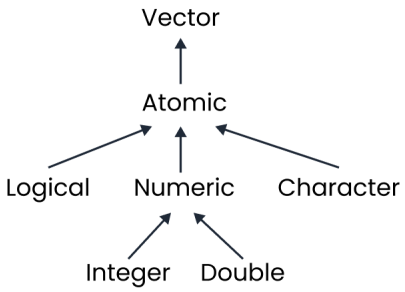


Figure 19.5: Atomic vectors

In our output `first_map_output`, we know that all elements in the list are numerics, specifically doubles.

```
first_map_output[[1]] |> typeof()
```

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

Since the list elements are all of the same type, we can create a homogeneous atomic double vector instead of a list. To accomplish this, we can use the `map_dbl` function.

```
(map_output_double <- c(1, 2, 3) |> map_dbl(.f = \(x) x^2))
```

```
[1] 1 4 9
```

Indeed, the output is a vector of doubles:

```
map_output_double |> typeof()
```

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

In contrast to `map`, the other functions in the map family create homogeneous atomic vectors. However, this can cause issues in practice. For example, if we want the `map_lgl` function to return a logical vector, and the input vector only contains the numbers 0 and 1:

```
c(0, 1, 1) |> map_lgl(.f = \(x) x^2)
```

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

In this case, it works as expected. As previously discussed, coercion rules convert the numbers 0 and 1 to `TRUE` and `FALSE`, and vice versa. However, `map_lgl()` doesn't allow for the coercion of numbers other than 0 and 1 to logicals.

```
c(0, 1, 2) |> map_lgl(.f = \(x) x^2)
```

```
Error in `map_lgl()`:
  In index: 3.
Caused by error:
! Can't coerce from a double vector to a logical vector.
Run `rlang::last_error()` to see where the error occurred.
```

An example of another error that is thrown because the coercion rules were not followed is this:

```
c(1, 2, 3) |> map_dbl(.f = \(x) list(x))
```

```
Error in `map_dbl()`:
  In index: 1.
Caused by error:
! Can't coerce from a list to a double vector.
Backtrace:
 1. purrr::map_dbl(c(1, 2, 3), .f = function(x) list(x))
 2. purrr:::map_("double", .x, .f, ..., .progress = .progress)
 3. purrr:::with_indexed_errors(...)
 4. base::withCallingHandlers(...)
```

This code doesn't work because `map_dbl` has no idea how to convert a list to a double (that's exactly what the error says if you look closely).

So, the key takeaway from these examples is to always be aware of what kind of output you're expecting when using the `map` functions. You will usually receive an error when you break the coercion rules of atomic vectors and lists. Which brings us to the next topic.

19.5 Using map with a list as input

In the examples shown before, the input was an atomic vector and the output was either an atomic vector or a list. However, this isn't always the case. Lists are another common input for the `map` functions. Let me show you an example:

```
list(x = 1, y = 2, z = 3) |> map_dbl(.f = \(x) x)
```

```
x y z
1 2 3
```

What's interesting about this example is that that we didn't have to subset the list element! In other words, `x` in the anonymous function `.f` represents the list element. Subsetting becomes necessary only when the input list contains list elements, as shown in this example:

```
list(
  x = list(1),
  y = list(4),
  z = list(4)
  ) |>
  map_dbl(.f = \(x) x[[1]])
```

```
x y z
1 4 4
```

Another important point to remember is that when we iterate over a list, we iterate through its individual elements. This holds true even if the elements themselves are more complex data structures like tibbles:

```
list(
  tibble1 = tibble(id = c(1, 2, 3), value = c(4, 5, 6)),
  tibble1 = tibble(id = c(1, 2, 3), value = c(8, 10, 33))
```

```
) |> map(
  .f = \(x) x$value
)
```

```
$tibble1
[1] 4 5 6

$tibble1
[1]  8 10 33
```

As the input list comprises of tibbles, `x` within the anonymous function refers to a tibble. Columns of tibbles can be accessed using the `$` operator, thus `x$value` returns the `value` column from each tibble within the list.

Working with lists and using the map function can become more challenging when each element in the list contains a different data structure. In this example, the first element is a tibble and the second an atomic vector of type numeric:

```
list(
  tibble1 = tibble(id = c(1, 2, 3), value = c(4, 5, 6)),
  vector1 = c(3, 4, 5)
) |> map(
  .f = \(x) {
    if (is_tibble(x)) (
      return(x$value)
    )

    x
  }
)
```

```
$tibble1
[1] 4 5 6

$vector1
[1] 3 4 5
```

The `map` function returns the `value` column from the tibble and `vector1` as well. To extract what is needed, I have used an if statement in the anonymous function's body.

Similarly, if statements can be used to avoid throwing errors. In this example, no error is thrown as the log of the character vector is not computed and instead the function returns an `NA`.

```
list(
  v1 = c(3, 4, 5),
  v2 = c(8, 22, 34),
  v3 = c("a", "b", "c")
) |> map(
  .f = \(x) {
      if(is.numeric(x)) (
        return(log(x))
      ) else if (is.character(x)) (
        return(NA)
      )
    }
  )
```

```
$v1
[1] 1.098612 1.386294 1.609438

$v2
[1] 2.079442 3.091042 3.526361

$v3
[1] NA
```

19.6 Error handling with safely and possibly

Error handling can be done better, however. The `safely` and `possibly` functions both deal with errors but handle them in different ways:

- `safely` creates a version of `.f` that always runs successfully. It returns a list with two elements: the result and the error. If the function call works, it returns the value, the error will be `NULL`. If an error occurs, the result will be `NULL` and error contains the error object.
- `possibly` modifies `.f` so that it returns a default value whenever an error occurs.

safely

Here's an example of using `safely`. The list contains two numerics and a character. The `map` function should calculate the log of each list element. Since you can't compute logs for characters, this function call would result in an error. But, if we wrap `log` in `safely`, `map` returns a list:

```
safe_log <- safely(log)

(log_res <- list(2, 3, "A") |>
  map(.f = \(x) safe_log(x)))
```

```
[[1]]
[[1]]$result
[1] 0.6931472

[[1]]$error
NULL


[[2]]
[[2]]$result
[1] 1.098612

[[2]]$error
NULL


[[3]]
[[3]]$result
NULL

[[3]]$error
<simpleError in .Primitive("log")(x, base): non-numeric argument to mathematical function>
```

Each element in the returned list has two elements: `result` and `error`. From the output, we can see two things. The third element under `result` shows that `map` computed `NULL` for `A`. Also, the third element contains an error object under `$error`.

At first glance, it may seem like you could use another `map` function to return only the result elements from the list, but it's more complicated than it appears, because this doesn't work in this example.

```
log_res |> map_dbl(.f = \(x) x$result)
```

```
Error in `map_dbl()`:
 In index: 3.
Caused by error:
! Result must be length 1, not 0.
Backtrace:
 1. purrr::map_dbl(log_res, .f = function(x) x$result)
 2. purrr:::map_("double", .x, .f, ..., .progress = .progress)
 3. purrr:::with_indexed_errors(...)
 4. base::withCallingHandlers(...)
```

Do you remember that we said that the length of the output of `map` will always be of the same length than the input? However, this is not the case here and that's why we get an error. The issue is that if you convert `NULL` to a number you get a numeric of length 0:

```
as.double(NULL) |> length()
```

```
[1] 0
```

To overcome this problem, we need to replace `NULL` with `NA`, since `NA` has a length of 1.

```
as.double(NA) |> length()
```

```
[1] 1
```

Luckily, `safely` has an argument called `otherwise` which allows you to assign a default value for elements that result in an error (the default value could also be a number, of course):

```
safe_log_na <- safely(log, otherwise = NA)

log_res_na <- list(2, 3, "A") |>
  map(.f = \(x) safe_log_na(x))

log_res_na |>
  map_dbl(.f = \(x) x$result)
```

```
[1] 0.6931472 1.0986123        NA
```

possibly

`possibly` takes a different approach than `safely`. Unlike `safely` it doesn't create the sub-elements `result` and `error`. Instead, it creates default values for elements that throw an error. In this example, whenever an error occurs, `.f` returns `NA`:

```
log_possibly <- possibly(log, otherwise = NA)

list(2, 3, "A") |>
  map_dbl(.f = \(x) log_possibly(x))
```

```
[1] 0.6931472 1.0986123        NA
```

19.7 map_vec

I've not been entirely honest with you. The `map` family of functions has another function called `map_vec`. It's a new function introduced in purrr 1.0.0. You might have noticed that I haven't discussed factors or dates so far. These data types are vectors, built on top of atomic vectors. `map_vec` is specifically designed for these types of vectors.

We can show that factors or dates are atomic vectors by checking their data type. Dates, for instance, are doubles in disguise:

```
as.Date('2022-01-31') |> typeof()
```

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

And factors are integers:

```
as.factor(c(1, 2, 3)) |> typeof()
```

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

Now, supppose you have created a vector of dates:

```
c(as.Date('2022-01-31'), as.Date('2022-10-12'))
```

```
[1] "2022-01-31" "2022-10-12"
```

You would like to increment each date in the vector by one month. For example, November 12, 2022 should become December 12, 2022. The lubridate package offers the %m+% operator and the period() function which in combination enable to increase dates:

```
library(lubridate)

as.Date('2022-11-12') %m+% months(1)
```

```
[1] "2022-12-12"
```

If you were to use `map` to increment each date in the vector, you would get a list as output, but not a date vector:

```
c(as.Date('2022-01-31'), as.Date('2022-10-12')) |>
  map(.f = \(x) x %m+% months(1))
```

```
[[1]]
[1] "2022-02-28"

[[2]]
[1] "2022-11-12"
```

The function `map_vec` was created so that you can iterate over datetime, date, and factor vectors and get the same type of vector in return. So, if we use `map_vec` instead of `map` in our example, the output of `map_vec` would be a date vector:

```
c(as.Date('2022-01-31'), as.Date('2022-10-12')) |>
  map_vec(\(x) x %m+% months(1))
```

```
[1] "2022-02-28" "2022-11-12"
```

Another useful application of `map_vec` is generating a series of dates within a specified range. For example, it can be used to create a vector of the same date for the next 50 years:

```
1:50 |>
  map_vec(\(x) as.Date(ISOdate(x + 2023, 11, 12)))
```

```
 [1] "2024-11-12" "2025-11-12" "2026-11-12" "2027-11-12" "2028-11-12"
 [6] "2029-11-12" "2030-11-12" "2031-11-12" "2032-11-12" "2033-11-12"
[11] "2034-11-12" "2035-11-12" "2036-11-12" "2037-11-12" "2038-11-12"
[16] "2039-11-12" "2040-11-12" "2041-11-12" "2042-11-12" "2043-11-12"
[21] "2044-11-12" "2045-11-12" "2046-11-12" "2047-11-12" "2048-11-12"
[26] "2049-11-12" "2050-11-12" "2051-11-12" "2052-11-12" "2053-11-12"
[31] "2054-11-12" "2055-11-12" "2056-11-12" "2057-11-12" "2058-11-12"
[36] "2059-11-12" "2060-11-12" "2061-11-12" "2062-11-12" "2063-11-12"
[41] "2064-11-12" "2065-11-12" "2066-11-12" "2067-11-12" "2068-11-12"
[46] "2069-11-12" "2070-11-12" "2071-11-12" "2072-11-12" "2073-11-12"
```

```
    # map_vec(\(x) ymd(paste(x + 2022, "-10", "-12"))) # alternative solution
```

19.8 Using map with nested data frames

A common use case for the `map` functions is creating new columns in nested data frames. Many beginners find it hard to work with nested data frames. But keep in mind that a nested column in a nested data frame is nothing but a list. In the following example, we have nested the `diamonds` data frame, excluding the `cut` of the diamonds:

```
(nested_cuts <- diamonds |>
  nest(data = -cut))
```

```
# A tibble: 5 x 2
  cut       data
  <ord>     <list>
1 Ideal     <tibble [21,551 x 9]>
2 Premium   <tibble [13,791 x 9]>
3 Good      <tibble [4,906 x 9]>
4 Very Good <tibble [12,082 x 9]>
5 Fair      <tibble [1,610 x 9]>
```

Have a look at the `data` column. It is a list. So everything we have learned in the chapter on lists can be applied when using the `mutate()` function in conjunction with `map()`. Before we go through an example, let's have a look at the general structure of combining `mutate()` and `map()`:

```
<DATAFRAME> |>
  mutate(
```

```
    <NEW_COLUMN_NAME> = map(<COLUMN_NAME>, .f = \(x) <FUNCTION_CALL>)
  )
```

It's crucial to remember that nested columns are treated as lists. Therefore, when iterating through a nested column, `<COLUMN_NAME>` represents a list and `x` represents the individual element of that list. When using the `mutate()` function in combination with `map()`, the function is applied to each row of the data frame, allowing iteration through each element of the list. The output of the `map` function will either be a list or an atomic vector, which will be the value for each corresponding row in the newly created column `<NEW_COLUMN_NAME>`.

In the context of the diamond example, if we were to use `map()` to iterate over the values in the `data` column and return only the individual element of that list, we would duplicate the `data` column to a new column called `data_copied`.

```
nested_cuts |>
  mutate(
    data_copied = map(data, .f = \(x) x)
  )
```

```
# A tibble: 5 x 3
  cut       data                  data_copied
  <ord>     <list>                <list>
1 Ideal     <tibble [21,551 x 9]> <tibble [21,551 x 9]>
2 Premium   <tibble [13,791 x 9]> <tibble [13,791 x 9]>
3 Good      <tibble [4,906 x 9]>  <tibble [4,906 x 9]>
4 Very Good <tibble [12,082 x 9]> <tibble [12,082 x 9]>
5 Fair      <tibble [1,610 x 9]>  <tibble [1,610 x 9]>
```

Instead of returning the tibble in `.f`, we can perform any operation on the tibble `x` that is possible with tibbles. For instance, we could extract the number of rows from each tibble and store them in a new column as an integer value:

```
nested_cuts |>
  mutate(
    n_rows = map_int(data, .f = \(x) nrow(x))
  )
```

```
# A tibble: 5 x 3
  cut       data                  n_rows
  <ord>     <list>                 <int>
1 Ideal     <tibble [21,551 x 9]>  21551
```

```
2 Premium   <tibble [13,791 x 9]>  13791
3 Good      <tibble [4,906 x 9]>    4906
4 Very Good <tibble [12,082 x 9]>  12082
5 Fair      <tibble [1,610 x 9]>    1610
```

A closer examination of the code will reveal that I used `map_int()` instead of `map()` because I knew the output would be an atomic vector of type integer. In this case, the `nrow()` function is being used to extract the number of rows from each tibble. However, it is also possible to chain multiple computations together using the pipe operator to transform tibbles and store them in a new column:

```
(nested_new_data <- nested_cuts |>
  mutate(
    new_data = map(data, .f = \(x)  x |>
                     filter(price > 8000) |>
                     select(depth, price, x, y, z)
                   )
  ))
```

```
# A tibble: 5 x 3
  cut       data                 new_data
  <ord>     <list>               <list>
1 Ideal     <tibble [21,551 x 9]> <tibble [2,684 x 5]>
2 Premium   <tibble [13,791 x 9]> <tibble [2,503 x 5]>
3 Good      <tibble [4,906 x 9]>  <tibble [572 x 5]>
4 Very Good <tibble [12,082 x 9]> <tibble [1,645 x 5]>
5 Fair      <tibble [1,610 x 9]>  <tibble [201 x 5]>
```

The following examples demonstrate the capabilities of working with nested columns using the purrr package. We will delve deeper into these techniques in a later chapter. One powerful feature is the ability to store plots within a column.

For instance, suppose you wanted to create a scatterplot from each tibble in the `data` column. Within each of these tibbles, the `x` column represents the length of the diamond in millimeters and the y column represents the width of the diamond in millimeters. These two columns can be plotted on the x- and y-axis, respectively, to create the scatterplot:

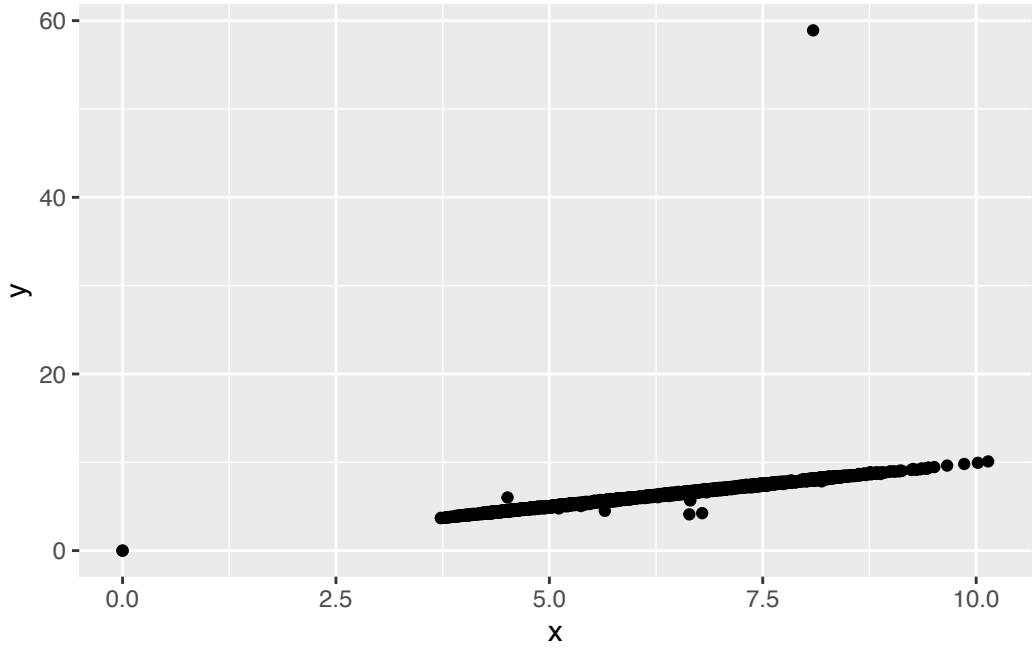
```
(plots <- nested_cuts |>
  mutate(
    scatterplots_x_y = map(data, .f = \(x) {
      x |>
        ggplot(aes(x = x, y = y)) +
```

```
        geom_point()
    })
  ))
```

```
# A tibble: 5 x 3
  cut       data                  scatterplots_x_y
  <ord>     <list>                <list>
1 Ideal     <tibble [21,551 x 9]> <gg>
2 Premium   <tibble [13,791 x 9]> <gg>
3 Good      <tibble [4,906 x 9]>  <gg>
4 Very Good <tibble [12,082 x 9]> <gg>
5 Fair      <tibble [1,610 x 9]>  <gg>
```

The tibble has been modified by the addition of a new column called `scatterplot_x_y`. Each value in this column contains a ggplot2 object, represented as `S3:gg`. The plots can be previewed by subsetting a specific value from the `scatterplot_x_y` column:

```
plots$scatterplots_x_y[[2]]
```



`S3:gg` objects are not the only type of objects that can be created using `map()` and `mutate()`. Another application of these two functions is fitting models to our data and storing the results

in a new column. For example, we could use `map()` and `mutate()` to fit a linear regression model to the `x` and `y` columns and store the model output in a new column:

```
(models <- nested_cuts |>
  mutate(
    fitted_model = map(data, \(x) {
      lm(x ~ y, data = x)
    })
  ))
```

```
# A tibble: 5 x 3
  cut       data                  fitted_model
  <ord>     <list>                <list>
1 Ideal     <tibble [21,551 x 9]> <lm>
2 Premium   <tibble [13,791 x 9]> <lm>
3 Good      <tibble [4,906 x 9]>  <lm>
4 Very Good <tibble [12,082 x 9]> <lm>
5 Fair      <tibble [1,610 x 9]>  <lm>
```

Models created with the function `lm()` are stored as `SE: lm` objects. For those with a background in statistics, it may be of interest to extract the effect size R^2 from these models. Within `map()`, we can call the `summary()` function on the `lm` object and access the `r.squared` attribute to extract the value of R^2.

```
models |>
  mutate(
    r_squared = map_dbl(fitted_model, \(x) summary(x)$r.squared)
  )
```

```
# A tibble: 5 x 4
  cut       data                  fitted_model r_squared
  <ord>     <list>                <list>           <dbl>
1 Ideal     <tibble [21,551 x 9]> <lm>             0.968
2 Premium   <tibble [13,791 x 9]> <lm>             0.880
3 Good      <tibble [4,906 x 9]>  <lm>             0.996
4 Very Good <tibble [12,082 x 9]> <lm>             0.998
5 Fair      <tibble [1,610 x 9]>  <lm>             0.988
```

We will delve much deeper into this concept in a later chapter. However, before we do so, it is important to become familiar with the more advanced `map2()` and `pmap()` functions, which will be covered in the next two chapters.

19.9 Using map with a list of data frames

There is a difference whether we iterate over the values of a column in a data frame, a single data frame, or multiple data frames. We have just seen the first case, where we iterate over the columns of a single data frame using `map()`. Now, let's discuss the last case, which is iterating over a list of data frames or tibbles. This is a common scenario when multiple data frames have been loaded into memory. To illustrate this, let's create a list containing three tibbles:

```
list_of_dframes <- list(
  d1 = tibble(
    d = c(1, 1, 1),
    p = c(34, 21, 9)
  ),
  d2 = tibble(
    d = c(2, 2, 2),
    p = c(21, 2, 76)
  ),
  d3 = tibble(
    d = c(3, 3, 3),
    p = c(54, 26, 11)
  )
)
```

Suppose you want to square the values in each `p` column. As before, we can use `map()` to perform this computation within the `.f` function.

```
list_of_dframes |>
  map(.f = \(dframe) {
    dframe |>
      mutate(p = p^2)
  })
```

```
$d1
# A tibble: 3 x 2
      d     p
  <dbl> <dbl>
1     1  1156
2     1   441
3     1    81

$d2
# A tibble: 3 x 2
```

```
      d     p
  <dbl> <dbl>
1     2   441
2     2     4
3     2  5776

$d3
# A tibble: 3 x 2
      d     p
  <dbl> <dbl>
1     3  2916
2     3   676
3     3   121
```

Now, suppose you want to combine these three data frames into one data frame. Prior to purrr 1.0.0, the function `map_dfr()` was used to bind the data frames automatically. However, with the release of purrr 1.0.0, the new function `list_rbind()` has been introduced, which combines data frames stored in a list.

```
list_of_dframes |>
  map(.f = \(dframe) {
    dframe |>
      mutate(p = p^2)
  }) |>
  list_rbind()
```

```
# A tibble: 9 x 2
      d     p
  <dbl> <dbl>
1     1  1156
2     1   441
3     1    81
4     2   441
5     2     4
6     2  5776
7     3  2916
8     3   676
9     3   121
```

Luckily, the `d` columns provides information about the origin of the data frames. This technique of binding data frames is particularly useful when multiple data frames are read into memory and need to be concatenated into a single data frame

Summary

- The `map` family of functions are used to iterate over the elements of atomic vectors, lists, and data frames and perform computations on them.
- Atomic vectors are homogeneous and one-dimensional data structures, while lists are heterogeneous, containing different types of data structures. Data frames are a specific type of list, where each element has the same length and data type.
- The `map` family of functions includes six functions: `map`, `map_lgl`, `map_int`, `map_dbl`, `map_chr`, and `map_vec`. The suffices `_lgl`, `_int`, `_dbl`, `_chr`, and `_vec` indicate the output data type of the function.
- When using one of the `map` functions in combination with `mutate` on a nested data frame, a nested column can be treated as a list.
- Nested data frames in combination with `mutate` and `map` are useful for creating plots or fitting models to the nested data.