Why use purrr::map instead of lapply?

Is there any reason why I should use

```
map(<list-like-object>, function(x) <do stuff>)
```

instead of

```
lapply(<list-like-object>, function(x) <do stuff>)
```

the output should be the same and the benchmarks I made seem to show that `lapply` is slightly faster (it should be as `map` needs to evaluate all the non-standard-evaluation input).

So is there any reason why for such simple cases I should actually consider switching to `purrr::map` ? I am not asking here about one's likes or dislikes about the syntax, other functionalities provided by purrr etc., but strictly about comparison of `purrr::map` with `lapply` assuming using the standard evaluation, i.e. `map(<list-like-object>, function(x) <do stuff>)` . Is there any advantage that `purrr::map` has in terms of performance, exception handling etc.? The comments below suggest that it does not, but maybe someone could elaborate a little bit more?

r purrr

edited Jul 17 '17 at 8:20

asked Jul 14 '17 at 10:45

Tim
2,517 1 13 28

4 For simple use cases indeed, better stick with base R and avoid dependencies. If you already load the `tidyverse` though, you may benefit from the pipe `%>%` and anonymous functions `~ .x + 1` syntax – Aurèle Jul 14 '17 at 10:53

26 This is pretty much a question of style. You should know what the base R functions do though, because all this tidyverse stuff is just a shell on top of it. At some point, that shell will break. – Hong Ooi Jul 14 '17 at 11:44

5 `~{}` shortcut lambda (with or without the `{}` seals the deal for me for plain `purrr::map()` . The type-enforcement of the `purrr::map_…()` are handy and less obtuse than `vapply()` . `purrr::map_df()` is a super expensive function but it also simplifies code. There's absolutely nothing wrong with sticking with base R `[lsv]apply()` , though. – hrbrmstr Jul 14 '17 at 13:49

14 Simple answer: Don't use it instead of `lapply` and just move on with your life- don't waste your time on trends/fashion nonsense. – David Arenburg Jul 15 '17 at 22:06

3 Thank you for the question - kind of stuff I also looked at. I am using R since more than 10 years and definitively don't and won't use `purrr` stuff. My point is following: `tidyverse` is fabulous for analyses/ interactive/reports stuff, not for programming. If you are into having to use `lapply` or `map` then you are programming and may end up one day with creating a package. Then the less dependencies the best. Plus: I sometime see people using `map` with quite obscure syntax after. And now that I see performances testing: if you are used to `apply` family: stick to it. – Eric Lecoutre Sep 1 '17 at 7:11

3 Answers

If the only function you're using from purrr is `map()` , then no, the advantages are not substantial. As Rich Pauloo points out, the main advantage of `map()` is the helpers which allow you to write compact code for common special cases:

- `~ . + 1` is equivalent to `function(x) x + 1`
- `list("x", 1)` is equivalent to `function(x) x[["x"]][[1]]` . These helpers are a bit more general than `[[` - see `?pluck` for details. For data rectangling, the `.default` argument is particularly helpful.

But most of the time you're not using a single `*apply()` / `map()` function, you're using a bunch of them, and the advantage of purrr is much greater consistency between the functions. For example:

- The first argument to `lapply()` is the data; the first argument to `mapply()` is the function. The first argument to all map functions is always the data.
- With `vapply()` , `sapply()` , and `mapply()` you can choose to suppress names on the output with `USE.NAMES = FALSE` ; but `lapply()` doesn't have that argument.
- There's no consistent way to pass consistent arguments on to the mapper function. Most functions use `...` but `mapply()` uses `MoreArgs` (which you'd expect to be called `MORE.ARGS`), and `Map()` , `Filter()` and `Reduce()` expect you to create a new anonymous function. In map functions, constant argument always come after the function name.
- Almost every purrr function is type stable: you can predict the output type exclusively from the

You may think that all of these minor distinctions are not important (just as some people think that there's no advantage to stringr over base R regular expressions), but in my experience they cause unnecessary friction when programming (the differing argument orders always used to trip me up), and they make functional programming techniques harder to learn because as well as the big ideas, you also have to learn a bunch of incidential details.

Purrr also fills in some handy map variants that are absent from base R:

- `modify()` preserves the type of the data using `[[<-` to modify "in place". In conjunction with the `_if` variant this allows for (IMO beautiful) code like `modify_if(df, is.factor, as.character)`
- `map2()` allows you to map simultaneously over `x` and `y`. This makes it easier to express ideas like `map2(models, datasets, predict)`
- `imap()` allows you to map simultaneously over `x` and its indices (either names or positions). This is makes it easy to (e.g) load all `csv` files in a directory, adding a `filename` column to each.

```
dir("\\.csv$") %>%
  set_names() %>%
  map(read.csv) %>%
  imap(~ transform(.x, filename = .y))
```

- `walk()` returns its input invisibly; and is useful when you're calling a function for its side-effects (i.e. writing files to disk).

Not to mention the other helpers like `safely()` and `partial()`.

Personally, I find that when I use purrr, I can write functional code with less friction and greater ease; it decreases the gap between thinking up an idea and implementing it. But your mileage may vary; there's no need to use purrr unless it actually helps you.

Microbenchmarks

Yes, `map()` is slightly slower than `lapply()`. But the cost of using `map()` or `lapply()` is driven by what you're mapping, not the overhead of performing the loop. The microbenchmark below suggests that the cost of `map()` compared to `lapply()` is around 40 ns per element, which seems unlikely to materially impact most R code.

```
library(purrr)
n <- 1e4
x <- 1:n
f <- function(x) NULL

mb <- microbenchmark::microbenchmark(
  lapply = lapply(x, f),
  map = map(x, f)
)
summary(mb, unit = "ns")$median / n
#> [1] 490.343 546.880
```

edited Nov 5 '17 at 15:48

answered Nov 5 '17 at 15:41

hadley
71.5k 20 142 203

1 Did you mean to use transform() in that example? As in base R transform(), or am I missing something? transform() gives you filename as a factor, which generates warnings when you (naturally) want to bind rows together. mutate() gives me the character column of filenames I want. Is there a reason not to use it there? – doctorG Nov 6 '17 at 15:49

1 Yes, better to use `mutate()`, I just wanted a simple example with no other deps. – hadley Nov 7 '17 at 2:13

Shouldn't type-specificity show up somewhere in this answer? `map_*` is what got me loading `purrr` in many scripts. It helped me with some 'control flow' aspects of my code (`stopifnot(is.data.frame(x))`). – Fr. Nov 13 '17 at 12:24

If we do not consider aspects of taste (otherwise this question should be closed) or syntax consistency, style etc, the answer is no, there's no special reason to use `map` instead of `lapply` or other variants of the apply family, such as the stricter `vapply`.

PS: To those people gratuitously downvoting, just remember the OP wrote:

> I am not asking here about one's likes or dislikes about the syntax, other functionalities provided by purrr etc., but strictly about comparison of purrr::map with lapply assuming using the standard evaluation

If you do not consider syntax nor other functionalities of `purrr`, there's no special reason to use `map`. I use `purrr` myself and I'm fine with Hadley's answer, but it ironically goes over the very things the OP stated upfront he was not asking.

edited Nov 6 '17 at 18:16

answered Jul 31 '17 at 22:47

Carlos Cinelli
6,093 4 25 54

This online purrr tutorial highlights the **convenience** of not having to explicitly write out anonymous functions when using purrr, which along with type-specific map functions makes it very functional.

1. purrr::map is syntactically much more convenient than lapply

extract second element of the list

```
map(list, 2)  # and it's done like magic
```

which as @F. Privé pointed out, is the same as:

```
map(list, function(x) x[[2]])
```

with lapply

```
lapply(list, 2) # doesn't work
```

we need to pass it the anonymous function

```
lapply(list, function(x) x[[2]])  # now it works
```

or as @RichScriven pointed out, we can simply pass `[[` as an argument into lapply

```
lapply(list, `[[`, 2)  # a bit more simple syntantically
```

In the background, purr takes either a numerical or character vector as an argument and uses that as a subsetting function. If you're doing lots and lots of subsetting of lists using lapply, and tire of either defining a custom function, or writing an anonymous function for subsetting, convenience is one reason to move to purrr.

2. Type-specific map functions simply many lines of code

- map_chr()
- map_lgl()
- map_int()
- map_dbl()
- map_df() - my favorite, returns a data frame.

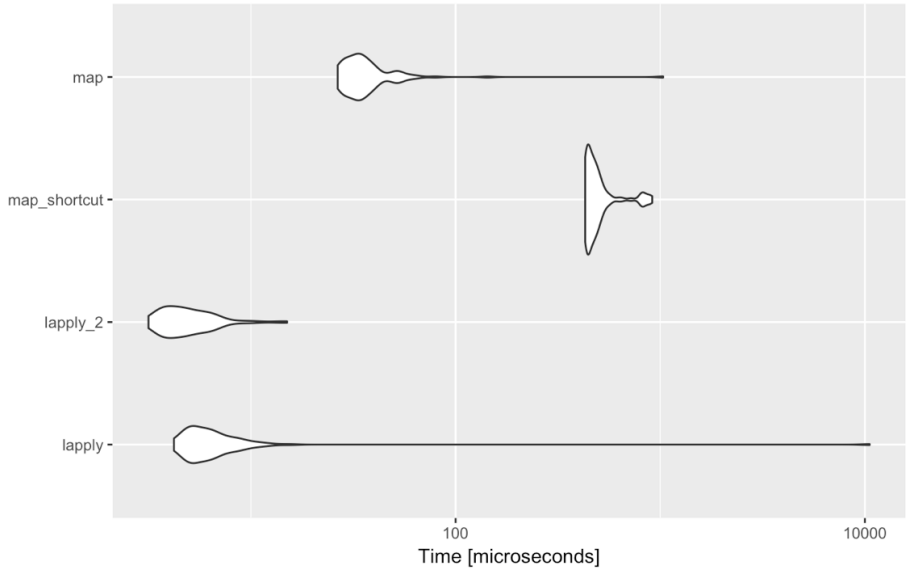
Each of these type-specific map functions returns an atomic list, rather than the list that `map()` and `lapply()` automatically return. If you're dealing with nested lists that have atomic vectors within, you can use these type-specific map functions to pull out the vectors directly, or coerce vectors up into int, dbl, chr vectors. Another point for convenience and functionality.

3. Convenience aside, lapply is faster than map.

Using the purrr convenience functions, as @F. Privé pointed out slows down processing a bit. Let's race each of the 4 cases I presented above.

```
# devtools::install_github("jennybc/repurrrsive")
library(repurrrsive)
library(purrr)
library(microbenchmark)
library(ggplot2)

mbm <- microbenchmark(
lapply = lapply(got_chars[1:4], function(x) x[[2]]),
lapply_2 = lapply(got_chars[1:4], `[[`, 2),
map_shortcut = map(got_chars[1:4], 2),
map = map(got_chars[1:4], function(x) x[[2]]),
times = 100
)
autoplot(mbm)
```



And the winner is....

```
lapply(list, `[[`, 2)
```

In sum, if speed is what you're after: **base::lapply**

If simple syntax is your jam: **purrr::map**

edited Feb 13 at 1:18

answered Sep 1 '17 at 6:31

Rich Pauloo
981 4 21

2 Note that if you use `function(x) x[[2]]` instead of just `2`, it would be less slow. All this extra time is due to checks that `lapply` doesn't do. – F. Privé Sep 1 '17 at 7:02

12 You don't "need" anonymous functions. `[[` is a function. You can do `lapply(list, "[[", 3)`. – Rich Scriven Sep 1 '17 at 7:02

@RichScriven that makes sense. That does simplify the syntax for using lapply over purrr. – Rich Pauloo Sep 1 '17 at 7:04