

Functional Programming

Using



July 23, 2022

Wolf Riepl - R Trainer
<https://statistik-dresden.de>

JOIN OUR NEXT WEBINAR

Functional Programming using purrr package

Date: July 23, 2022
Time: Saturday 10am (WAT)

Wolf Riepl
R Trainer
Workshops -
Data Analysis, Data Visualization, Data Mining

REGISTER NOW!
<https://bit.ly/3v3dssD>

Youtube: StatistikinDD
Twitter: @StatistikinDD

Let's bring on the Magic!



Source: giphy.com

We will use magic two-fold:

- The magic of the purrr package
- Magic numbers to generate messages from random letters



What We Want to Achieve

Iterate over Several Vectors in One R Command

```
purrr::pmap_chr(  
  list(seeds, choices, word_lengths),  
  magic_message) |>  
cat()
```

Happy Bday To You In 2022

Let's Go Step By Step

Drawing Random Letters ...

```
set.seed(1)
sample(letters, size = 5, replace = TRUE) |>
  paste(collapse = "")
```

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

```
set.seed(2)
sample(letters, size = 5, replace = TRUE) |>
  paste(collapse = "")
```

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

As Magicians, we should know some magic seeds ...

Knowing a Few Magic Seeds ...

```
set.seed(1982138)
sample(letters, size = 4,
      replace = TRUE) |>
  paste(collapse = "") |>
  stringr::str_to_title()
```

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

```
set.seed(942538)
sample(letters, size = 4,
      replace = TRUE) |>
  paste(collapse = "") |>
  stringr::str_to_title()
```

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

Is this code elegant,
especially when we create more words?

From Copy & Paste to Our Own Function

When to Write a Function

You should consider writing a function whenever you've copied and pasted a block of code more than twice (i. e. you now have three copies of the same code).

Hadley Wickham and
Garrett Grolemund
in **R for Data Science** (R4DS)

```
magic_message <- function(seed) {  
  set.seed(seed)  
  sample(letters,  
        size = 4,  
        replace = TRUE) |>  
  paste(collapse = "") |>  
  stringr::str_to_title()  
}
```

Applying the Function

```
magic_message(1982138)
```

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

```
magic_message(942538)
```

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

Applying the Function More Elegantly

```
seeds <- c(1982138, 942538)  
lapply(seeds, magic_message)
```

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

```
[[2]]  
[1] "Data"
```

```
sapply(seeds, magic_message)
```

```
[1] "More" "Data"
```

```
lapply(seeds, magic_message) |> unlist()
```

```
[1] "More" "Data"
```

Enter purrr

Specify Data Type of Return Value: `map_...()`



- `map()` corresponds to `lapply()`: returns a list
- `map_` variants return other data types
- `map_chr()`, `map_int()`, `map_dbl()`, `map_lgl()` etc.

For now, we will stick with `map_chr()`.

```
seeds <- c(1982138, 942538)
lapply(seeds, magic_message)
```

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

```
[[2]]
[1] "Data"
```

```
library(purrr)
map_chr(seeds, magic_message)
```

```
[1] "More" "Data"
```

purrr: Iterating over Two Vectors

Love is just a four-letter-word ...

So far, our `magic_message()` function can only return 4-letter-words:

```
magic_message <- function(seed) {  
  set.seed(seed)  
  sample(letters,  
    size = 4,  
    replace = TRUE) |>  
  paste(collapse = "") |>  
  stringr::str_to_title()  
}
```

What if word lengths differ?

Flexible word lengths

Let's change that:

```
magic_message <- function(seed,  
  word_length) {  
  set.seed(seed)  
  sample(letters,  
    size = word_length,  
    replace = TRUE) |>  
  paste(collapse = "") |>  
  stringr::str_to_title()  
}
```


purrr: Iterating over Two Vectors

Let's apply the new function

map2(.x, .y, .f, ...)

```
seeds <- c(2219868, 110454)
word_lengths <- c(5, 4)

map2_chr(seeds, word_lengths,
          magic_message)
```

```
[1] "Happy" "Bday"
```

What if we have more than 2 vectors to iterate over?

There is no **map3()** function in purrr

purrr: Iterating over Multiple Vectors

Adding Flexibility: Choices to Sample From

Remember the original *"magic message"*?

Happy Bday To You In 2022

So far, our `magic_message()` function can only output letters, not numbers.

Let's change that.

Flexible choices

```
magic_message <- function(seed,
  choices, word_length) {
  set.seed(seed)
  sample(choices,
    size = word_length,
    replace = TRUE) |>
  paste(collapse = "") |>
  stringr::str_to_title()
}
```

purrr: Iterating over Multiple Vectors

pmap(.l, .f, ...)

.l = list of vectors

```
seeds <- c(2219868, 110454, 639, 1750, 690, 9487)
word_lengths <- c(5, 4, 2, 3, 2, 4)
choices <- c(rep(list(letters), 5), list(0:9))
```

```
purrr::pmap_chr(
  list(seeds, choices, word_lengths),
  magic_message) |>
cat()
```

Happy Bday To You In 2022

```
purrr::pmap_chr(
  # .l list of vectors to iterate over
  list(seeds, choices, word_lengths),

  # .f function that takes 3 arguments
  magic_message) |>

# remove quotes around each word
cat()
```

OK, maybe wow, ...



Source: giphy.com

... but is it useful in a real-world application?

Example: Saving subsets of Data in separate csv files

using a *vector of filenames* and a *vector of categories* (subgroups)

See script: *Abuja-2-map2-csv.R*

Some Background about purrr



purrr: Basic Idea

`map(.x, .f, ...)`

Argument	Description	Example
<code>.x</code>	vector or list a data frame is a special case of a list	see <code>typeof(data)</code>
<code>.f</code>	function	<code>map(1:10, function(x) rnorm(10, x))</code>
<code>.f</code>	formula	<code>map(1:10, ~ rnorm(10, .x))</code>
<code>.f</code>	vector / list	<code>map(sw_films, "title")</code> extracts the title from sub-lists of <i>sw_films</i>
<code>...</code>	additional arguments passed to <code>.f</code>	

See Charlotte Wickham - An introduction to purrr: <https://github.com/cwickham/purrr-tutorial>

Base R: `apply()` vs. tidyverse: `purrr::map()`

```
apply(X, MARGIN, FUN, ...)  
lapply(X, FUN, ...)  
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)  
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)  
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)  
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)  
eapply(env, FUN, ..., all.names = FALSE, USE.NAMES = TRUE)
```

- Base R: Iterate over elements using `apply()` functions
- Inconsistent grammar and function arguments
- Hard to switch from one function to another

```
map(.x, .f, ...)  
map_if(.x, .p, .f, ...)  
map_at(.x, .at, .f, ...)  
map_lgl(.x, .f, ...)  
map_chr(.x, .f, ...)  
map_int(.x, .f, ...)  
map_dbl(.x, .f, ...)  
map_dfr(.x, .f, ..., .id = NULL)  
map_dfc(.x, .f, ...)
```

- `map()` functions: consistent grammar
- Easy to switch from one function to another

Source: <https://colinfay.me/happy-dev-purrr/>

purrr: map () Functions

purrr Function	Description	Example
map()	Applies a function to each element of the input; result is always a list (as in <code>lapply()</code>)	<code>map(1:4, rnorm, n = 5)</code> compare to <code>lapply(1:4, rnorm, n = 5)</code>
map_chr()	Result: <i>character vector</i> (string)	<code>map_chr(c("John F. Kennedy", "Barack Obama"), toupper)</code>
map_int()	Result: numeric <i>integer</i>	<code>map_int(c("John F. Kennedy", "Barack Obama"), nchar)</code>
map_dbl()	Result: numeric, including decimals (<i>double</i>)	<code>map_dbl(sample(1:10, 5), sqrt)</code>
map_lgl()	Result: <i>logical vector</i> (TRUE / FALSE)	<code>map_lgl(c(1, 2, NA, 4), is.na)</code>
map_dfr() / map_dfc()	Result: <i>data frame</i> , created by row binding / column binding	<code>map_dfr(c(a = 3, b = 10), rnorm, n = 100)</code>

Further *purrr* Functions

Function	Description
<code>walk2()</code>	analogous to <code>map2()</code> , no output in the console ("silent" return value <code>.x</code>)
<code>pwalk()</code>	analogous to <code>pmap()</code> , no output in the console
<code>partial()</code>	pre-fill function arguments, e. g.: <code>mean_na_rm <- partial(mean, na.rm = TRUE)</code> <code>mean_na_rm(c(1, 2, 3, NA))</code> <code>[1] 2</code>
<code>compact()</code>	Discard elements that are NULL or of length 0, e. g.: <code>list(a = 1:3, b = NULL, c = "text") %>% compact()</code>
<code>flatten()</code>	Remove a level of hierarchy from a list to make it less nested, e. g.: <code>rerun(2, sample(4)) %>% flatten_int()</code> compare to <code>rerun(2, sample(4))</code>
<code>safely()</code>	Modify a function: instead of aborting with an error, return a list with components <i>result</i> and <i>error</i> , e. g. <code>log_safe <- safely(log); log_safe("text")</code>
<code>possibly()</code>	Modify a function: instead of aborting with an error, return a default value (<i>otherwise</i>), e. g. <code>log_pos <- possibly(log, otherwise = "Doesn't make sense!"); log_pos("text")</code>

purrr: Learning More



RStudio CheatSheet <https://raw.githubusercontent.com/rstudio/cheatsheets/main/purrr.pdf>

R for Data Science (R4DS), Chapter 21: Iteration <https://r4ds.had.co.nz/>

Jenny Bryan, R Package *repurrrsive*:

- <https://github.com/jennybc/repurrrsive>
- <https://jennybc.github.io/purrr-tutorial/>

Charlotte Wickham

- <https://github.com/cwickham/purrr-tutorial>
- <https://www.rstudio.com/resources/rstudioconf-2017/happy-r-users-purrr-tutorial-/>

Colin Fay: Vous allez aimer avoir {purrr} (Article in English)

<https://colinfay.me/happy-dev-purrr/>

Now Apply Your Own Magic and Enjoy!



Wolf Riepl
Active on LinkedIn
<https://statistik-dresden.de/>

Youtube: StatistikinDD
Twitter: @StatistikInDD
<https://www.facebook.com/statistikdresden>