purrr beyond map()

a.k.a. fun with functional programming in R

```
result <- purrr::modify(.x = , .f = )
```

PREDICTIVE INSIGHTS

Hendrik van Broekhuizen
Predictive Insights
2020-03-07

hendrik@predictiveinsights.net
@hendrikvanb

The `purrr` package

what and why?

definition

> *A complete and consistent functional programming toolkit for R*
> *- `help(purrr)`*

objective

> *... to give you similar expressiveness to a classical FP language, while allowing you to write code that looks and feels like R*
> *- purrr 0.1.0*

`purrr`: what is it good for?

Absolutely ~~nothing~~ ~~everything~~ lots of stuff

Iterative tasks

- `lapply`++
  - more consistent, more general, more powerful

Working with lists

- Yes, even complex, nested lists
- It's lists, all the way down

Creating consistent, robust functions/routines

- Consistent syntax
- Fail loudly
- Nice error handling

Bringing some `tidyverse`-esque style to `data.table`

- `modify_if`, `modify_at`

The obligatory preamble

Making sure we are all on the same page

Disclaimer

- *`purrr` fantatic ¬□ `purrr` expert*
- 15min ≠ enough time

Admissions

- I'm an *extreme centrist* w.r.t. `tidyverse` and `data.table`
- I love to `%>%`
- I often find it useful to explicitly prefix functions with their namespace

Setup

- Working in RStudio in an `.Rproj` context
- Using same set of packages throughout

```r
library(data.table)
library(tidyverse)
library(lubridate)
```

Some tips

useful things to keep in mind when using purrr

When not to use `map()`

- `lapply()` is the base equivalent to `map()` (sans `purrr` *helpers* support)
  - if you're only using `map()` from purrr, you can skip the additional dependency and use `lapply()` directly
- there is no need to map if the operation is already appropriately vectorised

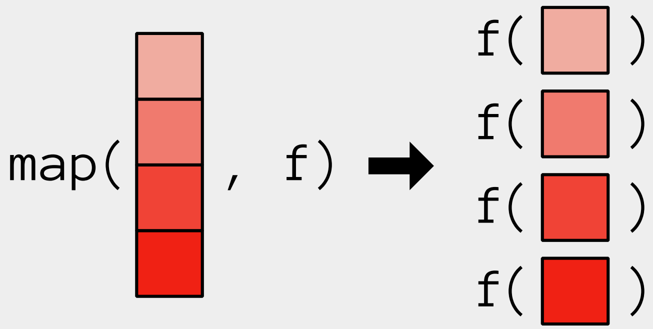
Avoiding nasty surprises

- `map*()` functions always return output of the same length as the input
- a data frame is simply a list of [consistently typed] vectors of equal length
  - lists vs atomic vectors vs vectors

An unsolicited piece of advice

- use inline anonymous functions instead of `purrr`'s formula syntax

A map() primer

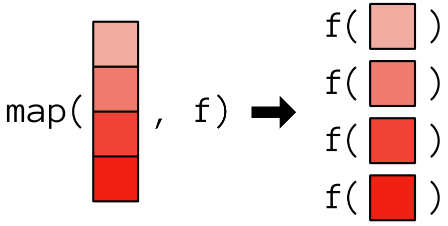


map()

Apply to all

`map(.x, .f)`

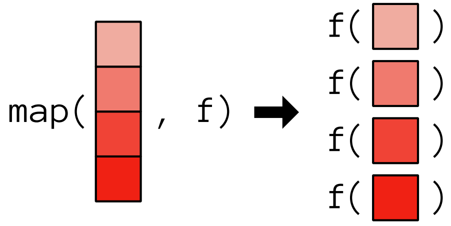
call function `.f` once for each element of vector `.x`;
return the result as a list



map()

Apply to all

`map(.x, .f)`
call function `.f` once for each element of vector `.x`; return the result as a list



Example:
Get the square of each number from 1 to 5

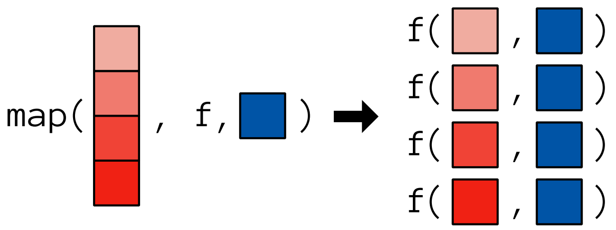
```r
# function to get square of number
my_square <- function(x) x^2

# get square of each number 1:5 and output as list
res1 <- 1:5 %>% map(my_square)                  # direct call
res2 <- 1:5 %>% map(~my_square(.))              # for backward compatibility
res3 <- 1:5 %>% map(~my_square(.x))             # formula
res4 <- 1:5 %>% map(function(x) my_square(x)) # inline anonymous function

# test equivalence
identical(res1, res2) & identical(res2, res3) & identical(res3, res4)
```

```
[1] TRUE
```

Passing arguments with . . .

Many ways to do to the same thing

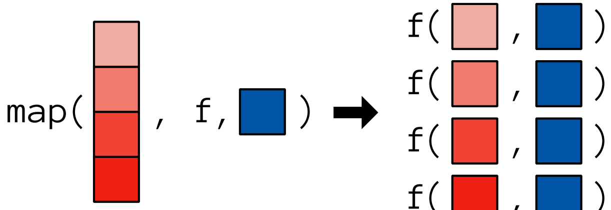
`map(.x, .f, ...)`
passes arguments specified in ... along



Passing arguments with ...

Many ways to do to the same thing

`map(.x, .f, ...)`
passes arguments specified in ... along



Example:
Use `paste()` to add 'min' as suffix to each number from 1 to 5

```
# pass arguments along
spec1 <- 1:5 %>% map(paste, 'min')

# formula specification (two variants)
spec2 <- 1:5 %>% map(~paste(., 'min'))
spec3 <- 1:5 %>% map(~paste(.x, 'min'))

# inline anonymous function specification
spec4 <- 1:5 %>% map(function(x) paste(x, 'min'))

# test equivalence
list(spec2, spec3, spec4) %>% map_lgl(identical, y = spec1)
```

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

Passing arguments: via `...` vs in function

A seemingly subtle, yet important difference

Not all that seems vecorised is...

- `map()` is only vectorised over its first argument so arguments passed to `map()` after `.f` will be
  - passed along as is and
  - evaluated once

What is that supposed to mean?

- Has implications if you pass arguments to function via `...`
  - errors if you pass vectors as arguments to functions that do not accept vectors as arguments
  - potentially wrong results even if arguments specified correctly

Passing arguments: via `...` vs in function

A seemingly subtle, yet important difference

Not all that seems vecorised is...

- `map()` is only vectorised over its first argument so arguments passed to `map()` after `.f` will be
    - passed along as is and
    - evaluated once

What is that supposed to mean?

- Has implications if you pass arguments to function via `...`
    - errors if you pass vectors as arguments to functions that do not accept vectors as arguments
    - potentially wrong results even if arguments specified correctly

E.g.:

```
# function that multiplies input (arg1) by specified constant (arg2)
temp_func <- function(x, constant = 2) {
  glue::glue('{x} x {constant} = {x*constant}')
}

# method 1: pass parameterised arg2 directly to map_chr
1:5 %>% map_chr(temp_func, constant = sample(1:10, 1))

# method 2: pass parameterised arg2 into inline anonymous function
1:5 %>% map_chr(function(x) temp_func(x, constant = sample(1:10, 1)))
```

```
[1] "1 x 2 = 2"  "2 x 2 = 4"  "3 x 2 = 6"  "4 x 2 = 8"  "5 x 2 = 10"
[1] "1 x 8 = 8"  "2 x 1 = 2"  "3 x 7 = 21" "4 x 7 = 28" "5 x 7 = 35"
```

map_*()

Dictacting the output format

`map_*(.x, .f, ...)`
call function `.f` once for each element of vector `.x`; return the result as an atomic vector of type `*`; error if impossible

- `map_chr(.x, .f)`: character
- `map_lgl(.x, .f)`: logical
- `map_dbl(.x, .f)`: real
- `map_int(.x, .f)`: integer
- `map_dfr(.x, .f)`: data frame (`bind_rows`)
- `map_dfc(.x, .f)`: data frame (`bind_cols`)

map_*()

Dictacting the output format

map_*(.x, .f, ...)
call function .f once for each element of vector .x; return the result as an atomic vector of type *; error if impossible

- map_chr(.x, .f): character
- map_lgl(.x, .f): logical
- map_dbl(.x, .f): real
- map_int(.x, .f): integer
- map_dfr(.x, .f): data frame (bind_rows)
- map_dfc(.x, .f): data frame (bind_cols)

Examples:

```
1:5 %>% map_chr(paste, 'min') %>% class()
```

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

```
1:5 %>% map_lgl(function(x) x < 3) %>% class()
```

```
[1] "logical"
```

```
1:5 %>% map_int(function(x) x * 2L) %>% class()
```

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

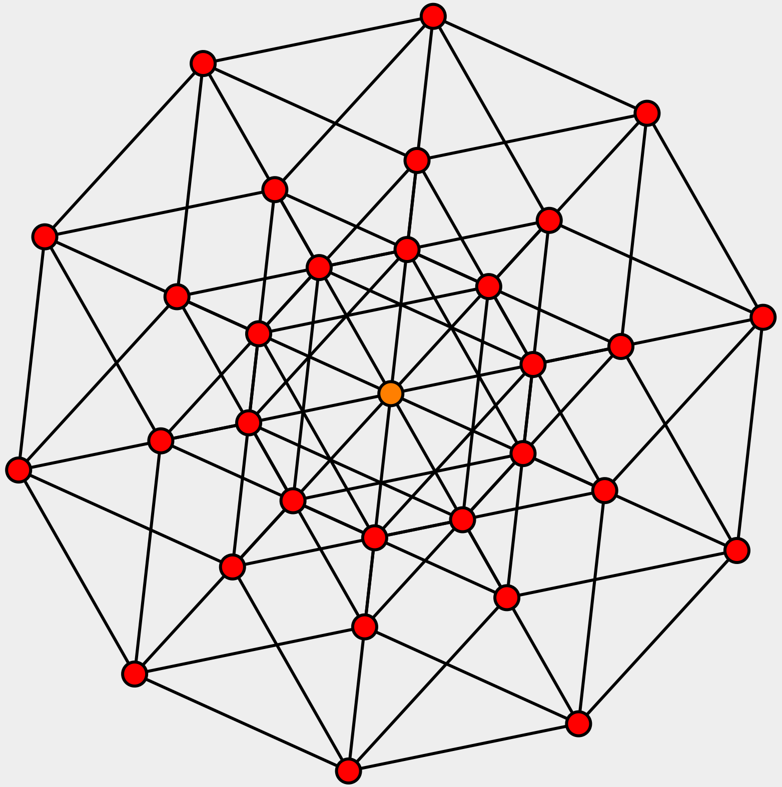
```
1:5 %>% map_df(function(x) data.frame(value = x)) %>% class()
```

```
[1] "data.frame"
```

```
1:5 %>% map_dfc(function(x) data.table(value = x)) %>% class()
```

```
[1] "data.table" "data.frame"
```

Map variants



`walk()` and `modify()`

`map()` has siblings...

`walk(.x, .f, ...)`

call function `.f` once for each element of `.x`; return nothing

`modify(.x, .f, ...)`

call function `.f` once for each element of `.x`; return the result as an object of the same type as `.x`

walk() and modify()

map() has siblings...

walk(.x, .f, ...)
call function .f once for each element of .x; return nothing

E.g.:

```
1:5 %>% purrr::walk(paste, 'min')
```

```
result <- 1:5 %>%
  purrr::walk(function(x) x ^ 2)

print(result)
```

```
[1] 1 2 3 4 5
```

modify(.x, .f, ...)
call function .f once for each element of .x; return the result as an object of the same type as .x

E.g.:

```
# obviously a character vector
x <- c('1', '2', '3', '4', '5')
```

```
# try to convert each element to integer using map_dbl
x %>% purrr::map_dbl(as.integer)
```

```
[1] 1 2 3 4 5
```

```
# try to convert each element to integer using modify
x %>% purrr::modify(as.integer)
```

```
[1] "1" "2" "3" "4" "5"
```

Why `walk()`? Why `modify()`?

what's the point?

> `walk(.x, .f, ...)`
> call function `.f` once for each element of `.x`; return nothing

Just do stuff

- Some functions just need to do stuff, not necessarily return stuff
    - E.g.: `cat()`, `message()`, `saveRDS()`, etc
- Particularly useful for disk I/O operations
- Allows input "passthrough"

> `modify(.x, .f, ...)`
> call function `.f` once for each element of `.x`; return the result as an object of the same type as `.x`

Change the content; keep the wrapper

- Some functions just need to change stuff, not necessarily create stuff
- Not everything needs to be coerced
    - What if input is already of the type we want as output?
    - Type preservation can be essential
- Particularly useful are the `modify_if()` and `modify_at()` variants

`map()` variants cheatsheet

Basic rules & the matrix of understanding

map variant rules:

1. `map()` returns listl `map_*()` returns vector of type specified
2. `modify()` returns same type as input
3. `walk()` returns nothing
4. Iterate over two inputs with `map2()`, `walk2()`, `modify2()`
5. Iterate over input and index with `imap()`, `imodify()`, `iwalk()`
6. Iterate over any number of inputs with `pmap()` and `pwalk()`

map variant matrix:

- map family of functions has orthogonal input and outputs
- can organise all the family into a matrix, with inputs in the rows and outputs in the columns

arguments	list	atomic	preserve type	nothing
one argument	`map()`	`map_lgl(), ...`	`modify()`	`walk()`
two arguments	`map2()`	`map2_lgl(), ...`	`modify2()`	`walk2()`
one argument + index	`imap()`	`imap_lgl(), ...`	`imodify()`	`iwalk()`
n arguments	`pmap()`	`pmap_lgl(), ...`	NA	`pwalk()`

`map()` & `map_*()`

Some practical illustrations and applications

Examples:

`scripts/map_family.R`

1. Try some specs
   use `map()` to fit multiple models to the same data
2. A bit of class
   use `map_*()` to iterate over columns of a tibble and extract each's class
3. The biggest year for hits
   use `map_*()` inside a tibble to create a new column from an existing list column
4. Forgetting the bad years
   use `map_*()` to filter a tibble based on a condition applied to a list column
5. Read it in; build it up
   use `map_dfr()` to build a sinlge tibble from multiple constituent csv files on disk
6. A decade of hits
   use `map_dfr()` to build a nested df by iterating over a vector and applying a parameterised function
7. Which spec is best?
   use `map_dfr()` to fit multiple models to the same data and build a tidy df with results

`modify()` & `walk()`

Some practical illustrations and applications

`modify()` examples:
`scripts/map_family.R`

1. Let's call a spade a spade
   conditionally modify a vector using `modify_if()`
2. Back to the future
   use `modify_at()` to target and modify specific vector elements
3. Double the hits, tripple the misses
   complex conditional modification of a vector/(list column) using `modify()`

`walk()` examples:
`scripts/map_family.R`

1. The mystery of the missing classes
   use `walk()` to iterate over columns of a tibble and output each's class

`*walk_*()`

Itteratively write data to disk using `purrr::pwalk()`

Example:
For each manufacturer in the `mpg` dataset, write a `.csv` file to disk containing only the data for that manufacturer

*walk_*()

Itteratively write data to disk using `purrr::pwalk()`

Example:
For each manufacturer in the `mpg` dataset, write a `.csv` file to disk containing only the data for that manufacturer

```
# check for files (show that there are none)
list.files('data/mpg')
```

```
character(0)
```

*walk_*()

Itteratively write data to disk using `purrr::pwalk()`

Example:
For each manufacturer in the `mpg` dataset, write a `.csv` file to disk containing only the data for that manufacturer

```
# check for files (show that there are none)
list.files('data/mpg')
```

```
character(0)
```

```
# create files by taking the mpg df %>% collapsing the data for each manufacturer into a list column %>% walking
over the two columns in the df and for each pair (i.e. row of manufacturer and data values) doing: {create path
variable to point to the path where the data should be written %>% write the data to disk in .csv format}
mpg %>%
  group_nest(manufacturer, keep = T) %>%
  purrr::pwalk(function(manufacturer, data) {
    path <- file.path('data/mpg', glue::glue('df_{manufacturer}.csv'))
    write_csv(data, path)
  })
```

*walk_*()

Itteratively write data to disk using `purrr::pwalk()`

Example:
For each manufacturer in the `mpg` dataset, write a `.csv` file to disk containing only the data for that manufacturer

```
# check for files (show that there are none)
list.files('data/mpg')
```

```
character(0)
```

```
# create files by taking the mpg df %>% collapsing the data for each manufacturer into a list column %>% walking
over the two columns in the df and for each pair (i.e. row of manufacturer and data values) doing: {create path
variable to point to the path where the data should be written %>% write the data to disk in .csv format}
mpg %>%
  group_nest(manufacturer, keep = T) %>%
  purrr::pwalk(function(manufacturer, data) {
    path <- file.path('data/mpg', glue::glue('df_{manufacturer}.csv'))
    write_csv(data, path)
  })
```

```
# check for files again (show that there are now files)
list.files('data/mpg') %>% {c(head(., 3), tail(., 3))}
```

```
[1] "df_audi.csv"       "df_chevrolet.csv"  "df_dodge.csv"      "df_subaru.csv"     "df_toyota.csv"
"df_volkswagen.csv"
```

map( ) variants

Some practical illustrations and applications

Examples:

`scripts/map_variants.R`

1. One year at a time
   use `pwalk()` to create objects from a tibble

I'm intrigued...

where can I learn more?

Reference (R/Rstudio)

- `help(package = purrr)`
- F1 to show function help
- F2 to inspect function

Reference (online)

- `purrr` cheatsheet
- `purrr` reference

Learning and understanding

- Jenny Bryan's purrr tutorial
- Hadley Wickham's Advanced R Chapter 9: Functionals