Introduction to `purrr`

Adventures with `map`

Colin Phillips
Data Science Lab

10 August 2018

References

- Jenny Bryan's `purrr` tutorial materials
  - Great primer on `purrr` - I borrowed a lot of material
- Official `tidyverse` website
  - Documentation and user guides

History

Rough history of `purrr`: base R

- In the beginning, there was the `apply` "family of functions":
  - `apply()`, `lapply()`, `sapply()`, `vapply()`, `mapply()`, `rapply()`, and `tapply()`
- Problem is, you don't know how to use them
- No, you don't
  - Can never quite get the syntax right first time for each variant
  - Outputs are not reliably in the right format

Rough history of `purrr`: `plyr`

- Along comes the `plyr` package:
  - `aaply`, `adply`, `alply`, `a_ply`, `daply`, `ddply`, `dlply`, `d_ply`, `laply`, `ldply`, `llply`, `l_ply`, `maply`, `mdply`, `mlply`, `m_ply`
- Some consistency!
  - First character is expected input format (array, dataframe, list, multiple lists)
  - Second character is expected output format (_ for nothing)
- Sometimes runs quite slowly
- No longer under active development
- Most useful function turned out to be ddply, which led to…

Rough history of `purrr`: `dplyr`

- `dplyr` can do almost everything you really wanted to do with `plyr`!
  - *If* all you really wanted was to manipulate dataframes
- What can't it do?
  - Replace the `apply` functionality. For that we need…

purrr: part of the core **tidyverse**

- The `map` "family of functions" in `purrr` is highly internally consistent
    - Learn once, use everywhere
- `purrr` functions always return exactly the right type of output
    - No more heisenbugs from inconsistent simplification
- A new syntax for anonymous functions makes life easier
    - Concise and hassle-free quick functions

```
library(tidyverse) # loads purrr (+ others)
```

```
## -- Attaching packages --------------------------------------------------------------------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 2.2.1     v purrr   0.2.4
## v tibble  1.4.2     v dplyr   0.7.4
## v tidyr   0.8.0     v stringr 1.3.0
## v readr   1.1.1     v forcats 0.3.0
```

```
## -- Conflicts ------------------------------------------------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Why `purrr`?

`purrr::map` vs `for` loops

- Is it possible to do everything that `purrr` can do, without `purrr`?
  - Yes, but
    - it’s ugly, error prone, and a waste of your time
- `for` loops can do everything that `apply` can do, just not as neatly
- `apply` functions can do everything that `purrr` can do, just not as neatly
- Code written with `purrr` functions are neater, cleaner, and have fewer bugs
  - [citation needed]

The `map` function

- We have a list: `.x`
- We have a function: `.f`
- We want to call the function `.f` on every element of the list `.x`
  - Remember that vectors and data frames are lists!
- Syntax: `map(.x, .f)`
- The `map` function will *always* return a list

```r
map(c(3,5,8), sqrt)
```

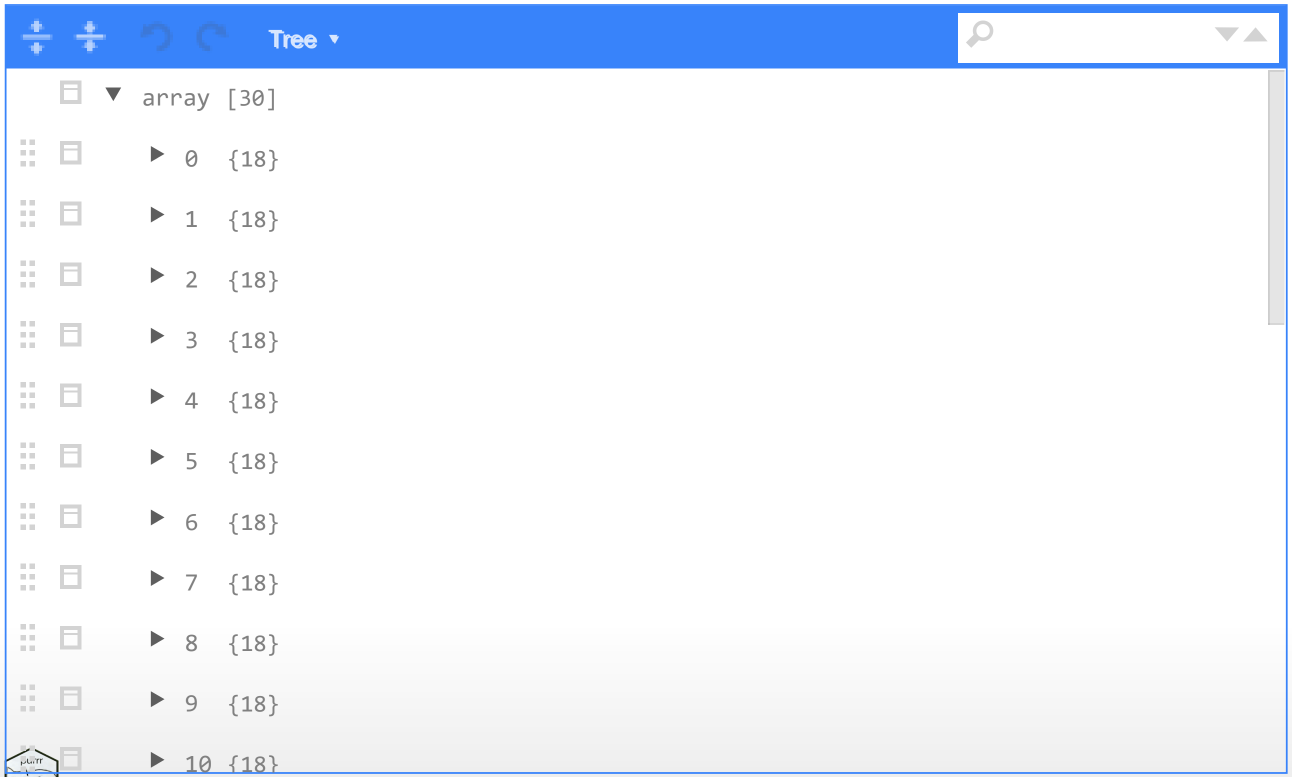
```
## [[1]]
## [1] 1.732051
##
## [[2]]
## [1] 2.236068
##
## [[3]]
## [1] 2.828427
```

Diving in

Example dataset

- Game of Thrones data
  - Originally from An API of Ice and Fire
  - Conveniently available offline in the `repurrrsive` package

```
library(repurrrsive, quietly = TRUE) # example datasets
library(listviewer) # to inspect lists interactively
jsonedit(got_chars) # let's look at the dataset
```



```
array [30]
  0 {18}
  1 {18}
  2 {18}
  3 {18}
  4 {18}
  5 {18}
  6 {18}
  7 {18}
  8 {18}
  9 {18}
  10 {18}
```

Calling a function on each element of the list

Each element of the list is independently passed in as the first argument to the function

```
# define a function that works on a single element of our list
count_titles = function(x) { length(x$titles) }
# this returns a list where each element is the count of titles that the character has
map(got_chars, count_titles)
```

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

Extracting by name

We can use the extract-by-name shortcut to pull out named elements of the *elements* of our list

```
# use the extractor shortcut
map(got_chars, "name")
```

```
## [[1]]
## [1] "Theon Greyjoy"
##
## [[2]]
## [1] "Tyrion Lannister"
##
## [[3]]
## [1] "Victarion Greyjoy"
##
## [[4]]
## [1] "Will"
##
## [[5]]
## [1] "Areo Hotah"
##
## [[6]]
## [1] "Chett"
```

Extract by position

We can use the extract-by-position shortcut to get the elements at a specific position in each of the elements in our list

```r
#use the extractor shortcut
map(got_chars, 3)
```

```
## [[1]]
## [1] "Theon Greyjoy"
##
## [[2]]
## [1] "Tyrion Lannister"
##
## [[3]]
## [1] "Victarion Greyjoy"
##
## [[4]]
## [1] "Will"
##
## [[5]]
## [1] "Areo Hotah"
##
## [[6]]
## [1] "Chett"
```

Note: `%>%` (pipe) syntax also works well

The pipe symbol (`%>%`) passes the left hand side as the first argument to the right hand side In this case, as the first argument to `map`

```r
# same as before
got_chars %>%
  map(3)
```

```
## [[1]]
## [1] "Theon Greyjoy"
##
## [[2]]
## [1] "Tyrion Lannister"
##
## [[3]]
## [1] "Victarion Greyjoy"
##
## [[4]]
## [1] "Will"
##
## [[5]]
## [1] "Areo Hotah"
##
## [[6]]
```

What if I don't want a list as output?

`purrr` provides *type-specific* `map` variants for targeting specific outut types. `purrr` does additional checks for you, and gives helpful error messages if there's a problem

- `map_chr`, `map_lgl`, `map_int`, `map_dbl`
- for *character*, *logical*, *integer*, *numeric* vectors

```
map_chr(got_chars[1:5], "name")
```

```
## [1] "Theon Greyjoy"     "Tyrion Lannister"  "Victarion Greyjoy"
## [4] "Will"              "Areo Hotah"
```

```
map_lgl(got_chars[1:5], "alive")
```

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

Alternatively

`flatten` removes the top level of hierarchy in the list.

```
map(got_chars[1:5], "name") %>% flatten_chr()
```

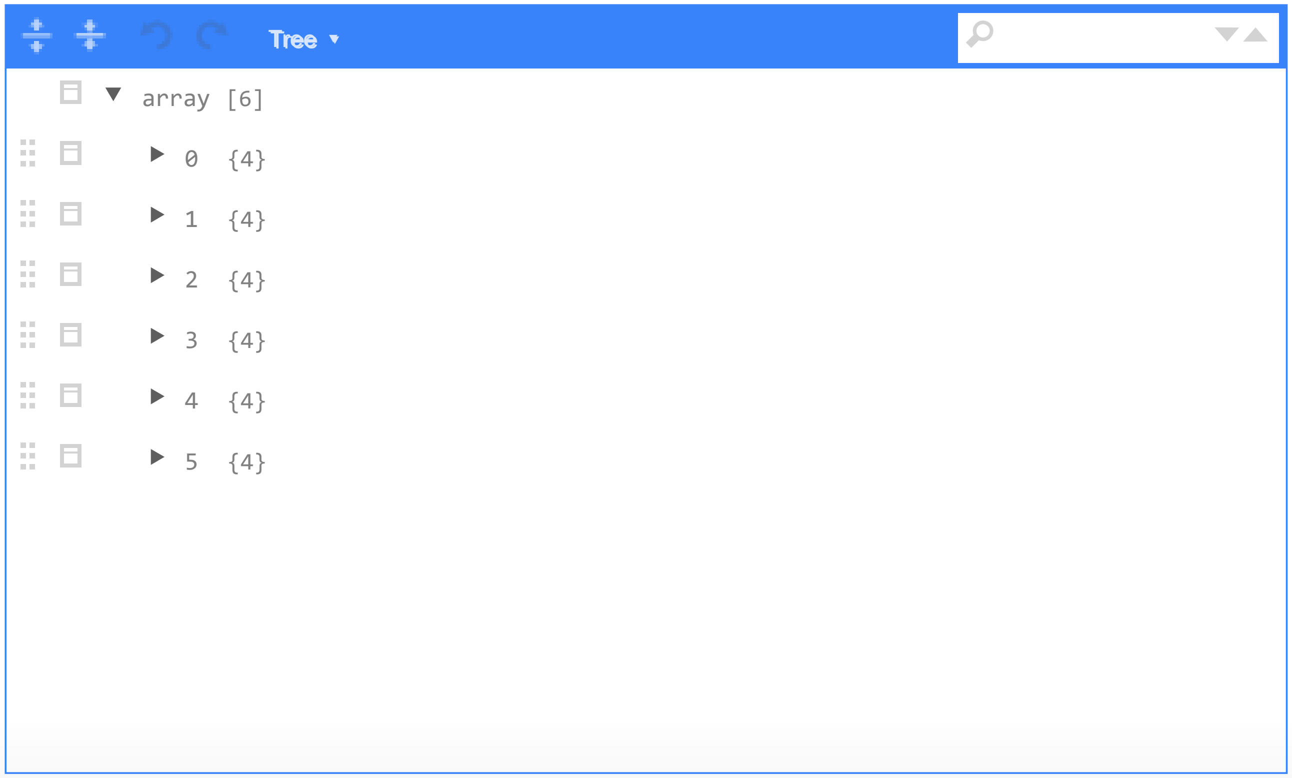
```
## [1] "Theon Greyjoy"     "Tyrion Lannister"  "Victarion Greyjoy"
## [4] "Will"              "Areo Hotah"
```

It's like base R's `unlist` but safer (fails out loud)

What if I want multiple outputs?

- `map` has a `...` argument to allow you to pass additional parameters to `.f`
- Helpful hack: pass the names you want as parameters to the "`[`" function

```
map(got_chars[3:8], `[`, c("name", "gender", "culture", "alive")) %>%
  jsonedit()
```



```
▼ array [6]
  ▶ 0 {4}
  ▶ 1 {4}
  ▶ 2 {4}
  ▶ 3 {4}
  ▶ 4 {4}
  ▶ 5 {4}
```

Can't I just have a data frame?

`map_df` - if the output list's elements will all be the same *length*

```
map_df(got_chars[3:8], `[`, c("name", "gender", "culture", "alive"))
```

```
## # A tibble: 6 x 4
##   name              gender culture  alive
##   <chr>             <chr>  <chr>    <lgl>
## 1 Victarion Greyjoy Male   Ironborn TRUE 
## 2 Will              Male   ""       FALSE
## 3 Areo Hotah        Male   Norvoshi TRUE 
## 4 Chett             Male   ""       FALSE
## 5 Cressen           Male   ""       FALSE
## 6 Arianne Martell   Female Dornish  TRUE
```

Notice that `map_df` automatically type converted the columns. Convenient, but not safe.

A safer, more tedious way

```
got_chars[3:8] %>% {
  tibble(
    name = map_chr(., "name"),
    gender = map_chr(., "gender"),
    culture = map_chr(., "culture"),
    alive = map_lgl(., "alive")
  )
}
```

```
## # A tibble: 6 x 4
##   name              gender culture  alive
##   <chr>             <chr>  <chr>    <lgl>
## 1 Victarion Greyjoy Male   Ironborn TRUE
## 2 Will              Male   ""       FALSE
## 3 Areo Hotah        Male   Norvoshi TRUE
## 4 Chett             Male   ""       FALSE
## 5 Cressen           Male   ""       FALSE
## 6 Arianne Martell   Female Dornish  TRUE
```

What if the data I need is nested?

A trick with the extractor shortcut - provide a list of how to proceed.

```
# for each character, get the first element of the povBooks vector for that character
got_chars %>%
  map_chr(list("povBooks", 1))
```

```
##  [1] "A Clash of Kings"     "A Game of Thrones"    "A Feast for Crows"
##  [4] "A Game of Thrones"    "A Feast for Crows"    "A Storm of Swords"
##  [7] "A Clash of Kings"     "A Feast for Crows"    "A Game of Thrones"
## [10] "A Clash of Kings"     "A Game of Thrones"    "A Feast for Crows"
## [13] "A Feast for Crows"    "A Dance with Dragons" "A Dance with Dragons"
## [16] "A Game of Thrones"    "A Feast for Crows"    "A Game of Thrones"
## [19] "A Feast for Crows"    "A Game of Thrones"    "A Storm of Swords"
## [22] "A Dance with Dragons" "A Game of Thrones"    "A Feast for Crows"
## [25] "A Dance with Dragons" "A Dance with Dragons" "A Storm of Swords"
## [28] "A Dance with Dragons" "A Storm of Swords"    "A Game of Thrones"
```

Different ways of specifying `.f`

Pre-define the function

```r
make_titles = function(x, collapse) { paste(x$titles, collapse = collapse) }
got_chars[1:3] %>%
  map_chr(make_titles, ", ")
```

```
## [1] "Prince of Winterfell, Captain of Sea Bitch, Lord of the Iron Islands (by law of the green lands)
## [2] "Acting Hand of the King (former), Master of Coin (former)"
## [3] "Lord Captain of the Iron Fleet, Master of the Iron Victory"
```

Different ways of specifying `.f`

Define an inline (anonymous) function

```r
got_chars[1:3] %>%
  map_chr(function(x, collapse) { paste(x$titles, collapse = collapse) }, ", ")
```

```
## [1] "Prince of Winterfell, Captain of Sea Bitch, Lord of the Iron Islands (by law of the green lands)
## [2] "Acting Hand of the King (former), Master of Coin (former)"
## [3] "Lord Captain of the Iron Fleet, Master of the Iron Victory"
```

Different ways of specifying `.f`

`purrr`'s formula syntax

```
got_chars[1:3] %>%
  map_chr( ~ paste(.x$titles, collapse = ", "))
```

```
## [1] "Prince of Winterfell, Captain of Sea Bitch, Lord of the Iron Islands (by law of the green lands)
## [2] "Acting Hand of the King (former), Master of Coin (former)"
## [3] "Lord Captain of the Iron Fleet, Master of the Iron Victory"
```

Use `.x` to refer to the data

Build a data frame from your list interactively

Use the `enframe` function, then add columns as necessary

```
got_chars[7:10] %>%
  set_names(map_chr(got_chars[7:10], "name", "value")) %>% # give each list element a name
  enframe() %>% # treat list as data frame
  mutate(born = map_chr(value, "born"),
         alive = map_lgl(value, "alive")) # use mutate to call map functions on list columns
```

```
## # A tibble: 4 x 4
##   name               value       born                                    alive
##   <chr>              <list>      <chr>                                   <lgl>
## 1 Cressen            <list [18]> In 219 AC or 220 AC                     FALSE
## 2 Arianne Martell    <list [18]> In 276 AC, at Sunspear                  TRUE
## 3 Daenerys Targaryen <list [18]> In 284 AC, at Dragonstone               TRUE
## 4 Davos Seaworth     <list [18]> In 260 AC or before, at King's Lan~ TRUE
```

We can also map over multiple lists together

`map2` (and `map2_dbl`, `map2_chr`, `map2_df` etc.) allows you to call a function on two lists in parallel

Syntax: `map2(.x, .y, .f, ...)`

```
got_names = got_chars %>% map("name")
got_born = got_chars %>% map("born")
map2_chr(got_names, got_born, ~ paste(.x, "was born", .y, sep = " "))
```

```
##  [1] "Theon Greyjoy was born In 278 AC or 279 AC, at Pyke"
##  [2] "Tyrion Lannister was born In 273 AC, at Casterly Rock"
##  [3] "Victarion Greyjoy was born In 268 AC or before, at Pyke"
##  [4] "Will was born "
##  [5] "Areo Hotah was born In 257 AC or before, at Norvos"
##  [6] "Chett was born At Hag's Mire"
##  [7] "Cressen was born In 219 AC or 220 AC"
##  [8] "Arianne Martell was born In 276 AC, at Sunspear"
##  [9] "Daenerys Targaryen was born In 284 AC, at Dragonstone"
## [10] "Davos Seaworth was born In 260 AC or before, at King's Landing"
## [11] "Arya Stark was born In 289 AC, at Winterfell"
## [12] "Arys Oakheart was born At Old Oak"
## [13] "Asha Greyjoy was born In 275 AC or 276 AC, at Pyke"
## [14] "Barristan Selmy was born In 237 AC"
```

What if we need to process many lists in parallel?

`pmap` takes a list of input lists Use `..1`, `..2`, `..3`, etc in formulae to refer to arguments, or predefine your function

```r
got_names = got_chars %>% map("name")
got_born = got_chars %>% map("born")
got_alive = got_chars %>% map("alive")
pmap_chr(list(got_names, got_born, got_alive), ~ paste(
  ..1, "was born",
  ..2, "and is",
  ifelse(..3, "still alive", "now dead"),
  sep = " "))
```

```
##  [1] "Theon Greyjoy was born In 278 AC or 279 AC, at Pyke and is still alive"
##  [2] "Tyrion Lannister was born In 273 AC, at Casterly Rock and is still alive"
##  [3] "Victarion Greyjoy was born In 268 AC or before, at Pyke and is still alive"
##  [4] "Will was born  and is now dead"
##  [5] "Areo Hotah was born In 257 AC or before, at Norvos and is still alive"
##  [6] "Chett was born At Hag's Mire and is now dead"
##  [7] "Cressen was born In 219 AC or 220 AC and is now dead"
##  [8] "Arianne Martell was born In 276 AC, at Sunspear and is still alive"
##  [9] "Daenerys Targaryen was born In 284 AC, at Dragonstone and is still alive"
## [10] "Davos Seaworth was born In 260 AC or before, at King's Landing and is still alive"
## [11] "Arya Stark was born In 289 AC, at Winterfell and is still alive"
```

Working with just part of the list

`map_at` and `map_if` allow you to specify a *predicate function* to decide which elements get processed. Saves a lot of typing!

```
1:5 %>% map_if(~ .x %% 2 == 0, function(z){ z/2}) %>% flatten_dbl()
```

```
## [1] 1 1 3 2 5
```

```
1:5 %>% map_at(c(1,3,5), ~ 3 * .x + 1) %>% flatten_dbl()
```

```
## [1]  4  2 10  4 16
```

Turning a list "inside out"

`transpose` turns a pair of lists into a list of pairs, and vice versa

```
pair_of_lists = list(a = list(1,2,3), b = list(4,5,6))
(list_of_pairs = transpose(pair_of_lists))
```

```
## [[1]]
## [[1]]$a
## [1] 1
##
## [[1]]$b
## [1] 4
##
##
## [[2]]
## [[2]]$a
## [1] 2
##
## [[2]]$b
## [1] 5
##
##
## [[3]]
## [[3]]$a
```

Turning a list "inside out"

`transpose` turns a pair of lists into a list of pairs, and vice versa

```
list_of_pairs = list(list(a = 1, b = 4),
                     list(a = 2, b = 5),
                     list(a = 3, b = 6))
(pair_of_lists = transpose(list_of_pairs))
```

```
## $a
## $a[[1]]
## [1] 1
##
## $a[[2]]
## [1] 2
##
## $a[[3]]
## [1] 3
##
##
## $b
## $b[[1]]
## [1] 4
##
## $b[[2]]
```

Other useful features

I haven't used these myself but they are good to know about

- `modify` (`modify_if`, `modify_at`, `modify_depth`) for making changes to list elements
- `walk` (`walk2`, `pwalk`) for calling a function for side-effects
- `imap` for working with both the list elements and the index of the elements
- `lmap` for working with sub-lists rather than list elements
- `accumulate` and `reduce` for summarising lists into a single value iteratively
- `invoke_map` for calling a list of functions, each with potentially different arguments

Recap

We've seen that `purrr` allows us to:

- Call functions on each element of an arbitrary list
- Extract single values at any point in the hierarchy by name or position
- Get type-safe outputs
- Define and call functions
  - On the every element of the list
  - On parts of the list
  - On multiple lists in parallel
- Transform the structure of a list

purrr