INTERMEDIATE FUNCTIONAL PROGRAMMING WITH PURRR

Functional programming in R

Colin Fay
Data Scientist & R Hacker at ThinkR

About computation in R

“To understand computations in R, two slogans are helpful:

- Everything that exists is an object.

- Everything that happens is a function call."

— John Chambers

```
class(`+`)
[1] "function"

class(`<-`)
[1] "function"
```

R as a functional programming language

Functions can be

- manipulated
- stored in a variable
- lambda
- stored in a list
- arguments of a function
- returned by a function

About "pure functions"

In a pure function:

- output only depends on input
- no "side-effect"

```
# Output depends only on inputs
# No side effect
sum(1:10)
[1] 55

mean(1:100)
[1] 50.5
```

Impure functions are useful

Impure functions:

- Depend on environment
- Have "side-effects"

```
# Outputs depends of environment
Sys.Date()
[1] "2018-10-04"

# Side effect only
write.csv(iris, "iris.csv")
```

Read more about functional programming

- *Advanced R, Functional programming*, H. Wickham
- *Functional Programming in R*, T. Mailund

INTERMEDIATE FUNCTIONAL PROGRAMMING WITH PURRR

Let's practice!

INTERMEDIATE FUNCTIONAL PROGRAMMING WITH PURRR

Tools for functional programming in purrr

Colin Fay
Data Scientist & R Hacker at ThinkR

High order functions

A high order function can:

- Take one or more functions as arguments
- Return a function

```
nop_na <- function(fun){
  function(...){
    fun(..., na.rm = TRUE)
  }
}

sd_no_na <- nop_na(sd)
sd_no_na( c(NA, 1, 2, NA) )
[1] 0.7071068
```

Three types of high order functions

- Functionals
- Function factories
- Function operators

Out In	Vector	Function
Vector		Function factory
Function	Functional	Function operator

Advanced R, Functional Programming

Adverbs in purrr

Handling errors and warnings:

- `possibly()`
- `safely()`

```
library(purrr)
safe_mean <- safely(mean)
class(safe_mean)
[1] "function"
```

Use safely() to handle error.

`safely()` returns a function that will return:

- `$result`
- `$error`

```
safe_log <- safely(log)

safe_log("a")

$result
NULL

$error
<simpleError in log(x = x, base = base):
non-numeric argument to mathematical function>
```

Use safely() to handle error

```
map( list(2, "a"), log )

Error in log(x = x, base = base) :
  non-numeric argument to mathematical function
```

```
map( list(2, "a"), safely(log) )

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

[[1]]$error
NULL


[[2]]
[[2]]$result
NULL

[[2]]$error
<simpleError in log(x = x, base = base):
    non-numeric argument to mathematical function>
```

Extracting elements from safely() results

`map()` & `"result"` or `"error"`

```
safe_log <- safely(log)

map( list("a", 2),  safe_log) %>%
  map("result")

[[1]]
NULL

[[2]]
[1] 0.6931472
```

```
safe_log <- safely(log)

map( list("a", 2), safe_log ) %>%
  map("error")

[[1]]
<simpleError in log(x = x,
base = base): non-numeric argument
to mathematical function>

[[2]]
NULL
```

INTERMEDIATE FUNCTIONAL PROGRAMMING WITH PURRR

Let's practice!

INTERMEDIATE FUNCTIONAL PROGRAMMING WITH PURRR

possibly()

Colin Fay
Data Scientist & R Hacker at ThinkR

About possibly()

`possibly() creates a function that returns either:

- the result
- the value of `otherwise`

```
library(purrr)

possible_sum <- possibly(sum, otherwise = "nop")

possible_sum(1)
[1] 0

possible_sum("a")
[1] "nop"
```

Using possibly()

`possibly()` can return:

- A logical

```
ps <- possibly(sum, FALSE)
ps("a")
[1] FALSE
```

- A `NA`

```
ps <- possibly(sum, NA)
ps("a")
[1] NA
```

- A character

```
ps <- possibly(sum, "nope")
ps("a")
[1] "nope"
```

- A number

```
ps <- possibly(sum, 0)
ps("a")
[1] 0
```

INTERMEDIATE FUNCTIONAL PROGRAMMING WITH PURRR

Let's practice!

INTERMEDIATE FUNCTIONAL PROGRAMMING WITH PURRR

Handling adverb results

Colin Fay
Data Scientist & R Hacker at ThinkR

Cleaning safely results

Transform the result with `transpose()` :

```
# Transpose turn a list of n elements a and b
# to a list of a and b, with each n elements
l <- list("a", 2, 3)

map(l, safe_log) %>% length()

[1] 3

map(l, safe_log) %>% transpose() %>% length()

[1] 2
```

About compact()

`compact()` removes the `NULL`:

```
list(1, NULL, 3, 4, NULL) %>%
  compact()

[[1]]
[1] 1

[[2]]
[1] 3

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

possibly() and compact()

`otherwise = NULL %>% compact():`

```
l <- list(1,2,3,"a")

possible_log <- possibly(log, otherwise = NULL)

map(l, possible_log) %>% compact()

[[1]]
[1] 0

[[2]]
[1] 0.6931472

[[3]]
[1] 1.098612
```

A Gentle introduction to httr

- httr: a friendly http package for R

  H. Wickham

- Getting started with httr

  H. Wickham

INTERMEDIATE FUNCTIONAL PROGRAMMING WITH PURRR

Let's practice!