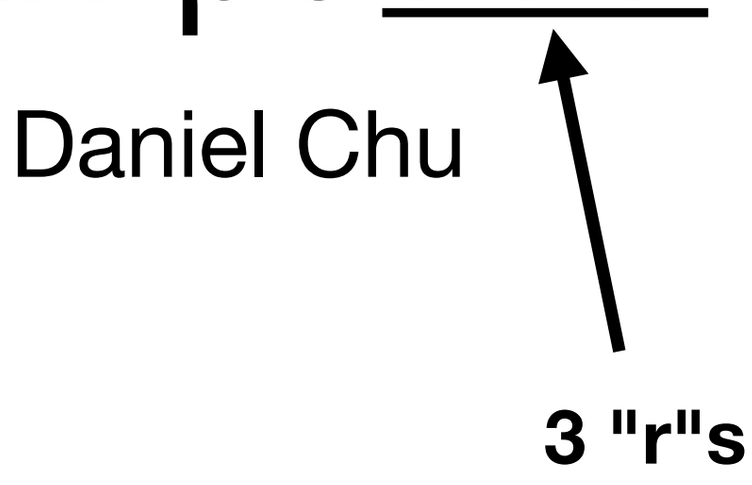
Functional programming with purrr

Daniel Chu



Functional Programming

Overly simplified definition:

1. Treats programming as the evaluation of statements (like mathematical statements).

2. Inputs to a function remain unchanged when a function is evaluated.

3. Running a function multiple times with the same inputs will return the same results.

4. Functions can take in functions as arguments and/or return functions as results (higher-order functions).

Higher-order Functions

```
blarg <- function(x) {
  (x + 1)^2
}
```

```
blurg <- function(x) {
  x(1:10)
}
```

```
> blurg(blarg)
 [1]   4   9  16  25  36  49  64  81 100 121
> blurg(mean)
[1] 5.5
```

Higher-order Functions

```
zlarg <- function(base) {
  if (missing(base)) stop("Hammer time.")
  function(x) log(x, base = base)
}
```

```
> (zlorg <- zlarg(3))
function(x) log(x, base = base)
<environment: 0x3b02970e0>

> zlorg(27)
[1] 3
```

Higher-order Functions

```
launch <- function(x, f, ...) {
  f(x, ...)
}
```

```
> zig <- rpois(1e6, 5)

> launch(zig, mean)
[1] 5.002474

> launch(zig, var)
[1] 5.010907
```

base

```
> Reduce(function(x, y) x + y, 1:5)
[1] 15

> is_div5 <- function(x) x %% 5 == 0

> Filter(is_div5, 4:14)
[1]  5 10

> Find(is_div5, 4:14)
[1] 5

> Find(Negate(is_div5), 4:14)
[1] 4
```

base::lapply

```
> lapply(3:4, sample.int, replace = TRUE)
[[1]]
[1] 1 2 1

[[2]]
[1] 2 1 3 2

> blag <- c(rnorm(1e6), NA)
> lapply(list(mean, sd),
+        function(x) x(blag, na.rm = TRUE))
[[1]]
[1] 0.0007692677

[[2]]
[1] 0.999884
```

base::Map

```
> flurg <- function(x, y, z) x + y - z

> unlist(Map(flurg,
+                x = 1:10,
+                y = 10:1,
+                z = 6))
 [1] 5 5 5 5 5 5 5 5 5 5
```

`purrr` Un-difficults Things

Makes life easier by:

- Ensures consistent results
- Integrates expectations in results
- Uses an easy shorthand notation
- Uses consistent language across functions

Results in:

- Less work.
- Fewer surprises.

purrr::map/map_*

```
# 1. Like base::lapply, always returns a list:
map(1:10, runif)

# 2. Assess if the number is even. Must return a
logical vector:
map_lgl(1:10, function(x) x %% 2 == 0)

# 3. Using shorthand notation for (2).
map_lgl(1:10, ~ . %% 2 == 0)

# 4. This also works as well as (3).
map_lgl(1:10, ~ .x %% 2 == 0)
```

purrr::map/map_*

```
# purrr::map works as an extractor function when given
a character, logical, or numerical vector instead of a
function.

list_df <-
  map(1:10, ~ sample_n(mtcars, 10, replace = TRUE))

# 1. extracts the "mpg" column from each data frame in
the list, returns a list:
map(list_df, "mpg")

# 2. Same as (1) but calls dplyr::bind_cols afterwards:
map_dfc(list_df, "mpg")
```

purrr::map_if/at

```
# purr::map_if: Use or generate a logical vector to
selectively apply a given function. Returns a list.

# Only embiggen even numbers by a factor of two:
map_if(1:10,
       ~ . %% 2 == 0,
       ~ . * 2)

# purr::map_at: Use or generate a vector of positions to
selectively apply a given function. Returns a list.

# Embiggen everything except the first three values:
map_at(1:10,
       -(1:3),
       ~ . * 2)
```

purrr::map2

```
# 1. Iterate over two vectors, returns a
list:
map2(1:10, 11:20, function(x, y) x + y)


# 2. Same as (1), must return a vector of
integers:
map2_int(1:10, 11:20,
         function(x, y) x + y)

# 3. Shorthand notation of (2):
map2_int(1:10, 11:20, ~ .x + .y)
```

purrr::pmap/pmap_*

```
# 1. Iterate over multiple vectors, returns a list:
pmap(list(x = 1:10, y = 10:1, z = 6),
     function(x, y, z) x + y - z)

# 2. Same as (1) but must return a vector of
doubles:
pmap_dbl(list(x = 1:10, y = 10:1, z = 6),
         function(x, y, z) x + y - z)

# 3. Same as (1) but must return a vector of
integers:
pmap_int(list(x = 1:10, y = 10:1, z = 6L),
         function(x, y, z) x + y - z)
```

purrr::pmap/pmap_*

```
# 4. Name-matched arguments can be out of order:
pmap_dbl(list(x = 1:10, z = 6, y = 10:1),
         function(x, y, z) x + y - z)

# 5. Shorthand notation of (2), arguments used in
order:
pmap_dbl(list(x = 1:10, y = 10:1, z = 6),
         ~ ..1 + ..2 - ..3)

# 6. Same as (5):
pmap_dbl(list(1:10, 10:1, 6),
         ~ ..1 + ..2 - ..3)
```

purrr::safely

```
# purrr::safely "wraps" a given function so that the error
side effects are captured and do not interrupt a process.

# Evaluation interrupted with an error:
log(c("4", 4))

# Not interrupted, no results:
safely(log)(c("4", 4))

# Interrupted:
map(list("4", 4, "four"), log)

# Not interrupted and result generated for the cromulent
value:
map(list("4", 4), safely(log))
```

Parallel Computing

`furrr` provides drop-in parallelized replacements of `purrr` map functions.

```
> library(tictoc)
> library(purrr)
> library(furrr) # packages future + purrr = furrr, GET IT?!
> plan(multiprocess) # 4 logical cores

> data <- data.frame(n = 1e4, size = 1e4, p = runif(1e4))

> tic(); x <- pmap(data, rbinom); toc()
6.606 sec elapsed

> tic(); y <- future_pmap(data, rbinom); toc()
5.876 sec elapsed
```

Lifting

```
# purrr::lift/lift_*: "lifts" the domain of a function.

# Does not work:
pmap_dbl(mtcars, sd)

# Does work, lift input argument(s) from vector to dots:
pmap_dbl(mtcars, lift_vd(sd))

# Also works, but with more typing and uncertainty:
pmap_dbl(mtcars, function(...) sd(c(...)))

# A more useful example:
sd_mlurg <-
  select(zablorkian_df, starts_with("mlurg_obs_")) %>%
  pmap_dbl(lift_vd(sd))
```

Other Useful Stuff

```
# purrr::reduce: Just like base::Reduce but with a bold new
flavor.

reduce(1:5, ~ .x + .y)
reduce(1:5, `+`) # also works

# purrr::partial: Partially fill in arguments of a function.

read.csv_new_1 <- function(x, ...) {
  read.csv(x, ..., stringsAsFactors = FALSE)
}

# same, but different
read.csv_new_2 <- partial(read.csv,
                          stringsAsFactors = FALSE)
```

Other Useful Stuff

```
# purrr::insistently: Modifies a function to retry in a
given amount of time if it encounters an error.

hammer_the_server <- insistently(google_api_call_function)

google_data <- hammer_the_server(args)

# purrr::slowly: Modifies a function to wait between each
call.

caress_the_server <- slowly(google_api_call_function,
                            rate = rate_delay(60))

glarg <- map(glarg_args, caress_the_server)
```