CHARLOTTE WICKHAM

HAPPY R USERS PURRR: USING FUNCTIONAL PROGRAMMING TO SOLVE ITERATION PROBLEMS

WARMUPS

YOUR TURN

```
pets <- list(
  scylla = list(type = "cat", colour = "calico"),
  dexter = list(type = "cat", colour = "black")
)
```

What is the difference between `pets["scylla"]` and `pets[["scylla"]]`?

How would you extract the colour of Dexter?

Reminder for Charlotte:

```
getElement(pets, "scylla")
```

```
listviewer::jsonedit(pets, mode = "view")
```

YOUR TURN

```
count_na <- function(x, ...){
  sum(is.na(x), ...)
}
```

What are the arguments to this function?

What does the function return?

What happens with the . . . ?

FUNCTIONAL PROGRAMMING

a programming paradigm

has some central concepts

you don't need to know them to use purrr, but I'll point them out

FP

USING FUNCTIONAL PROGRAMMING TO SOLVE ITERATION PROBLEMS

FOR EVERY ____ DO ____

You are already solving them:

copy & paste, for loops, `(l/s)apply()`

I'll show you an alternative `purrr::map()` & friends

```
df$a <- (df$a - min(df$a, na.rm = TRUE)) /
           (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))

df$b <- (df$b - min(df$b, na.rm = TRUE)) /
           (max(df$a, na.rm = TRUE) - min(df$b, na.rm = TRUE))

df$c <- (df$c - min(df$c, na.rm = TRUE)) /
           (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))

df$d <- (df$d - min(df$d, na.rm = TRUE)) /
           (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

DISCUSS WITH YOUR NEIGHBOUR:

1. What does this code do?
2. What are the sources of repetition?

ROADMAP

1. Functions
2. Iteration
3. Basic purrr
4. More purrr

GOAL:

Code that is easier to
write,
understand, and
update.

FUNCTIONS

IF YOU HAVE COPY-PASTED TWICE, IT'S TIME TO WRITE A FUNCTION.

Hadley Wickham

HOW TO WRITE A FUNCTION

Don't start with `fun_name <- function() {}`!

1. Start with a simple concrete case and solve the problem
2. Refer to inputs using temporary names (these will become argument names)
3. Simplify for understanding and remove duplication
4. Finally, wrap into a function, choosing a good name

LIVE CODE Write a function for rescaling

```
rescale01 <- function(x){
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

WRITE A FUNCTION THAT GIVEN A VECTOR, REMOVES THE LAST ELEMENT.

PURE FUNCTIONS & SIDE EFFECTS

FP

Pure functions:

- the result depends on nothing outside the function arguments
- changes nothing in the outside world (i.e. has no side effects)

Pure functions are easy to reason about

But side effects are integral to doing data analysis

Separate computations and side effects into different functions

DISCUSS WITH YOUR NEIGHBOUR

Which of the following functions have **side effects**:

1. `mean()`
2. `plot()`
3. `lm()`
4. `write.csv()`

ITERATION

```
l <- list(a = rnorm(5),
          b = rnorm(5),
          c = rnorm(5),
          d = rnorm(5))

l$a <- rescale01(l$a)
l$b <- rescale01(l$b)
l$c <- rescale01(l$c)
l$d <- rescale01(l$d)
```

REMOVE REPETITION BY WRITING A FOR LOOP

What would you change if don't want to store the results in the original object?

Hint:
Think of `l$a` as `l[[1]]`

LIVE CODE Let's do that to another list, and another.

```
rescale_cols <- function(x){
  output <- vector("list", length(x))
  for(i in seq_along(x)){
    output[[i]] <- rescale01(x[[i]])
  }
  output
}
```

WRITE A NEW FUNCTION

Instead of rescaling each column, it finds the mean of each column.

LIVE CODE What about taking the range of each column?

FUNCTIONS CAN BE ARGUMENTS

```
compute_cols <- function(x, fun){
  output <- vector("list", length(x))
  for(i in seq_along(x)){
    output[[i]] <- fun(x[[i]])
  }
  output
}

compute_cols(l, rescale01)
compute_cols(l, mean)
compute_cols(l, range)
```

A FUNCTION THAT WRITES FOR LOOPS FOR US!

Abstracts aways the details of writing a for loop,
so we can concentrate on the details of what is happening.

"for each column, rescale to [0,1]"

`purrr::map(l, rescale01)`

FUNCTIONS CAN BE ARGUMENTS

Functions are **first class** citizens in R, they can occur anywhere a number could:

- as arguments,
- as return values,
- assigning them to variables,
- storing in data structures.

Higher order function: a function that takes a function as input or returns a function

FP

WHAT ABOUT `sapply()` & `lapply()`?

What type of object does `sapply()` return?

Motivation for `purrr`:

- consistent return type,
- useful shortcuts,
- consistent syntax for more complicated iteration

PURRR BASICS

to each element of **.x** apply **.f**

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

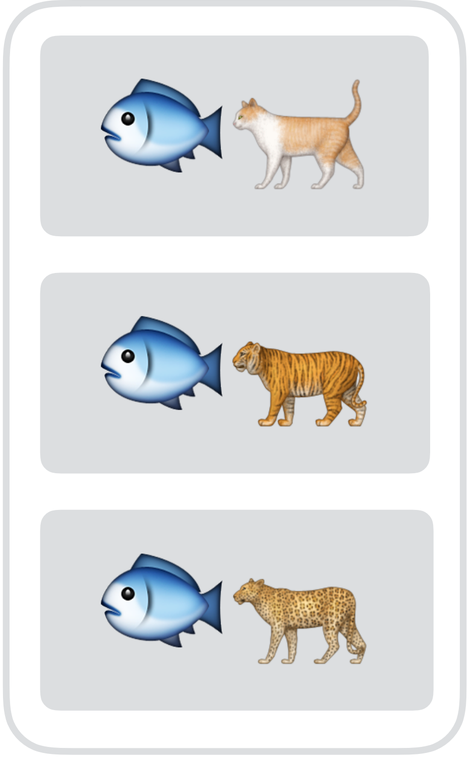
```
.f(.x[[1]], ...)

.f(.x[[2]], ...)

.f(.x[[3]], ...)
```

and so on

under the hood, "like" our `compute_col()` function

to each `cat` apply `give_fish`

```
map(  , give_fish )
```



INPUT IS ALWAYS A VECTOR

`.x` has to be a **vector**

- atomic vectors (e.g. logical, integer etc.), `x[[i]]` - i^{th} entry
- lists, `x[[i]]` - contents of the ith element
- data.frames, `x[[i]]` - contents of the ith column

YOUR TURN

Star Wars API - http://swapi.co/
Data extracted using https://github.com/Ironholds/rwars/

```
# loads objects: films, people, vehicles, starships, planets & species

load("data/swapi.rda")

map(films, length)

map(films, names)

map(films, getElement, "title")

map(films, getElement, "characters")
```

WHAT KIND OF OBJECT COMES OUT OF `map()`?

FOR SIMPLER OUTPUT

`map_chr()` - returns a **character** vector

`map_lgl()` - returns a **logical** vector

`map_int()` - returns a **integer** vector

`map_dbl()` - returns a **double** vector

`walk()` - when you want nothing at all, more later...

Result: a vector **same length as input** or **ERROR!**

Compare:

```
map(films, length)

map_int(films, length)
```

YOUR TURN

- Get the titles of films as a character vector
- Get the number of characters in each film
- Get the number of crew in each vehicle as an integer vector

Hint: if you get an error from a `map_*()` function, roll back to `map()` and figure out why

EXTRACTION SHORTCUTS

`map(films, getElement, "characters")` This pattern is so common,

it gets a shortcut:

`map(films, "characters")`

Extraction shortcut:

`.f` string or **number**, say `y`, then converted to the function `function(x) x[["y"]]`

SPECIFYING .F

`.f` can be:

- the **name** of a defined function (your's or someone else's)
- an **anonymous function** FP
- a **formula,** `~ .x` equivalent to `f(x) x`

LIVE CODE Demonstrate shortcuts

SHORTCUTS

.f		for each x in xs
string	`map(xs, "y")`	`x[["y"]]`
numeric	`map(xs, 1)`	`x[[1]]`
function	`map(xs, g)`	`g(x)`
formula	`map(xs, ~ g(.x))`	`g(x)`
anonymous function	`map(xs, function(x) g(x))`	`g(x)`

LIVE CODE Find the first name for all people

FIND THE BODY MASS INDEX OF ALL THE PEOPLE IN THE STAR WARS FILMS

pipe operator

```
x %>% f(y)
```

is equivalent to

```
f(x, y)
```

a common pattern:

```
map(x, f) %>% map(g)
```

for each x, apply f, then for each of the results, apply g

LIVE CODE Find the number of eye colors in each species

YOUR TURN Find the number of characters in each file using a series of maps

SO FAR...

`purrr`:

- writes for loops for you
- has type-consistent output
- has shortcuts to make code concise

Next up: more iteration patterns `map2()`, `pmap()`, `invoke_map()`

IF YOU HAVE COPY-PASTED TWICE,
IT'S TIME TO WRITE A FUNCTION.

Hadley Wickham

TRY TO REDUCE REPETITION WITH purrr:map(),
IF .f IS UNWIELDY, OR YOU USE THE SAME .f IN A LOT OF PLACES,
WRITE A FUNCTION

OTHER ITERATION FUNCTIONS

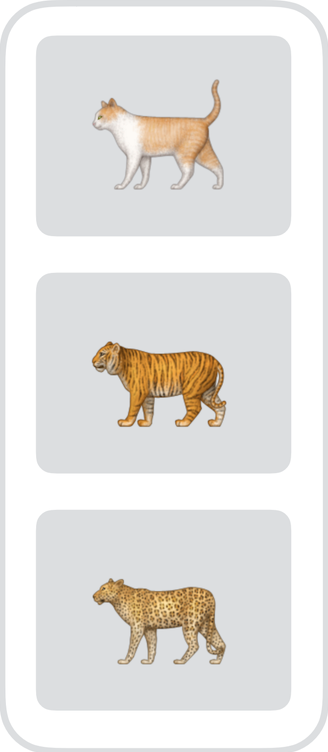
YOUR TURN

```
sampling_dist_mean <- function(n, pop_dist, n_sim = 1000, ...){
  colMeans(matrix(pop_dist(n = n * n_sim, ...), ncol = n_sim))
}

sampling_dist_mean(5, pop_dist = rnorm)
hist(sampling_dist_mean(5, pop_dist = rnorm))
```

USE `map()` TO GENERATE SAMPLING DISTRIBUTIONS FOR SAMPLES OF SIZE 2, 5, 10 AND 25?

LIVE CODE `walk()` to generate histograms

```
walk(
```



```
, love)
```

Expect nothing in return

You actually get .x invisibly back,
good for piping

For functions called for their side effects:

- printing to screen
- plotting to graphics device
- file manipulation (saving, writing, moving etc.)
- system calls

to each element of **.x** and corresponding element of **.y** apply **.f**

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

```
.f(.x[[1]], .y[[1]], ...)

.f(.x[[2]], .y[[2]], ...)

.f(.x[[3]], .y[[3]], ...)
```

and so on

to each `cat` and corresponding `times` apply `rep`

```
dists <- c(rnorm, rexp, rnorm, rexp)
ns <- c(2, 5, 10, 25)

sims2 <- map2(ns, dists, sampling_dist_mean)
```

FIX TO PRODUCE ALL COMBINATIONS OF SAMPLE SIZE AND POPULATION DISTRIBUTION

LIVE CODE `walk2()` to generate labelled histograms

CHALLENGES:

`challenges/01-mtcars.R` - Fit and summarise many regression models

`challenges/02-word_count.R` - Count the number of words of all files in a directory

`challenges/03-starwars.R` - Print who used which vehicles in the films

`challenges/04-weather.R` - Download, tidy, plot and save daily temperatures

`challenges/05-swapi.R` - Download all Star Wars data using `rwars` package

Next up: a few remaining iteration functions, a comment about other functions in purrr, wrap up.

to each element of each vector in .l, apply .f

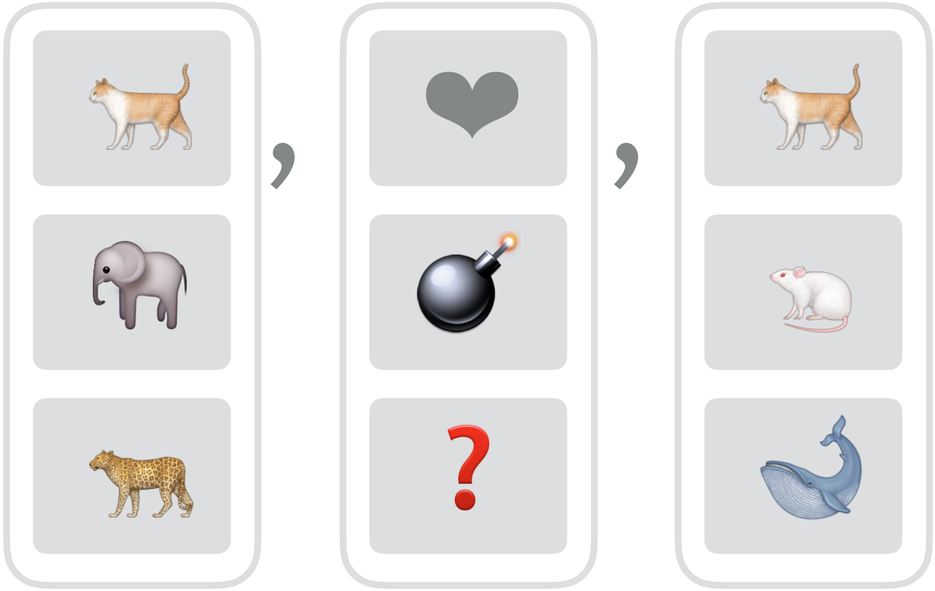
pmap(.l , .f , ...)

```
.f(.l[[1]][[1]], .l[[2]][[1]], .l[[3]][[1]], ...)

.f(.l[[1]][[2]], .l[[2]][[2]], .l[[3]][[1]], ...)

.f(.l[[1]][[3]], .l[[2]][[3]], .l[[3]][[1]], ...)
```

and so on

or by name if supplied

to each element `in animal, reaction, and animal2`, apply `c`

pmap(data.frame(), c)

for each function in **.f**, apply it to **.x**

`invoke_map(.f, .x, ...)`

```
.f[[1]](.x, ...)

.f[[2]](.x, ...)

.f[[3]](.x, ...)
```

and so on

for each function in **.f**, apply it to **.x**

```
invoke_map(give_fish, 🐈)
```

```
double
count_legs
```

🐟🐈

🐈🐈

4

PURRR AND LIST COLUMNS

```
mtcars %>%
  nest(-cyl) %>%
  mutate(
    fit = map(data, ~ lm(mpg ~ disp, data =.x)),
    slope = map_dbl(fit, ~ coef(.x)[2])
)
```

a list column

a list column

`map` - if you want another list column

`map_*` - if you want a "normal" column

OTHER FEATURES OF PURRR

WORKING WITH LISTS AND FUNCTIONS PURRR

Key objects in `purrr`: lists, functions

`purrr` provides functions to make working with functions and lists easier

Cheatsheet? (work in progress)

Example: `safely()`, `transpose()`

YOUR TURN

Look at the help for `safely()` and `transpose()`

What kind of objects do they expect as input?

What kind of objects are returned as output?

LIVE CODE Putting them together to handle errors

APPLICATIONS / CHALLENGES

Reading in many files

Working with file structures

Working with hierarchical data (e.g. JSON, xml…)

Working with many datasets or models (or use list columns + dplyr)

LEARNING MORE

R for Data Science

- http://r4ds.had.co.nz/functions.html
- http://r4ds.had.co.nz/iteration.html

DataCamp Writing functions in R
https://www.datacamp.com/courses/writing-functions-in-r

Jenny Bryan's purrr tutorial
https://github.com/jennybc/purrr-tutorial

THANK YOU

Slides and code @

@cvwickham

cwickham@gmail.com