Purrrrrrrrrrrrrrrrrrrrrrrrrrrr (tidyverse iterations)

Reminder

```
a <- 0
while (a < 100){
   a <- a+1
}
```

Base R loop

```
library(purrr)
map_dbl(1:100, function(a){a+1})
```

purrr iteration C++ loop

```
a <- 1:100
a <- a + 1
```

core vector operation (C loop)

```
a <- 0
for (i in 1:10){
   a <- a+1
}
```

Base R loop

Why purrr?
What is functional programming?

- "*purrr enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors*"
  - purrr's overview (https://purrr.tidyverse.org/)
- What makes a programming language "*functional*"?
  - Functions behave like any other data structure (e.g. you can "pass them" as a variable)
  - Functions are *pure*
    - Output depends only on input (consistent), i.e., runit(), read_csv(), Sys.time() are not pure
    - No side-effects (e.g., not changing global variables or writing to a file)
    - Hence, R is not entirely "*functional*" but adopts some elements

A functional

- Functional is a function that takes a function as an input and returns a vector as an output (like in math), e.g.:

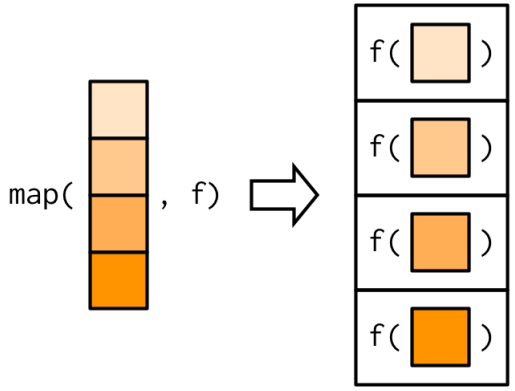
```
randomize <- function(f) f(runif(1e3))

randomize(mean)

randomize(sd)

randomize(plot)

integrate(sin, lower = 0, upper = pi/2)

map, lapply, apply,…
```

Be specific

- As a “best practice”, when you are using iterations, you should be as specific as possible, and pre-allocate as much as possible.
- In other words your preference should be:
  - purrr > for > while > repeat

purrr::map

- Takes a vector and a function and does this:
- The **implementation is in C code**,
- but is equivalent logically to something like:




```
simple_map <- function(x, f, ...) {
   out <- vector("list", length(x))
   for (i in seq_along(x)) {
      out[[i]] <- f(x[[i]], ...)
   }
   out
}
```

- map() returns a list, map_dbl(), map_lgl(), map_int(), map_chr() return a vecor of type double, logical, integer, and character.

A moment to think

- When you run map(iris, mean) what would you get?
- What would you get with map_dbl(iris, mean)
- Explain the output using the “equivalent loop”:

```
simple_map <- function(x, f, ...) {
   out <- vector("list", length(x))
   for (i in seq_along(x)) {
      out[[i]] <- f(x[[i]], ...)
   }
   out
}
```

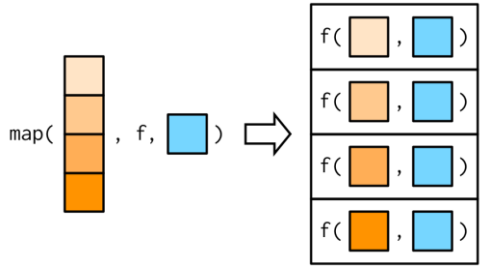
4 minutes

Defining functions for the use of map/map_*

- Inline anonymous function:
  - `map_dbl(iris, function(x) length(unique(x)))`
- Formula ~ shortcut
  - `map_dbl(iris, ~length(unique(.x)))`
- Or define it then use it:
  - `my_func <- function(x) { length(unique(x)) }`
  - `map_dbl(iris, my_func)`

Passing additional arguments to map

```
x <- list(1:5, c(1:10, NA))
```



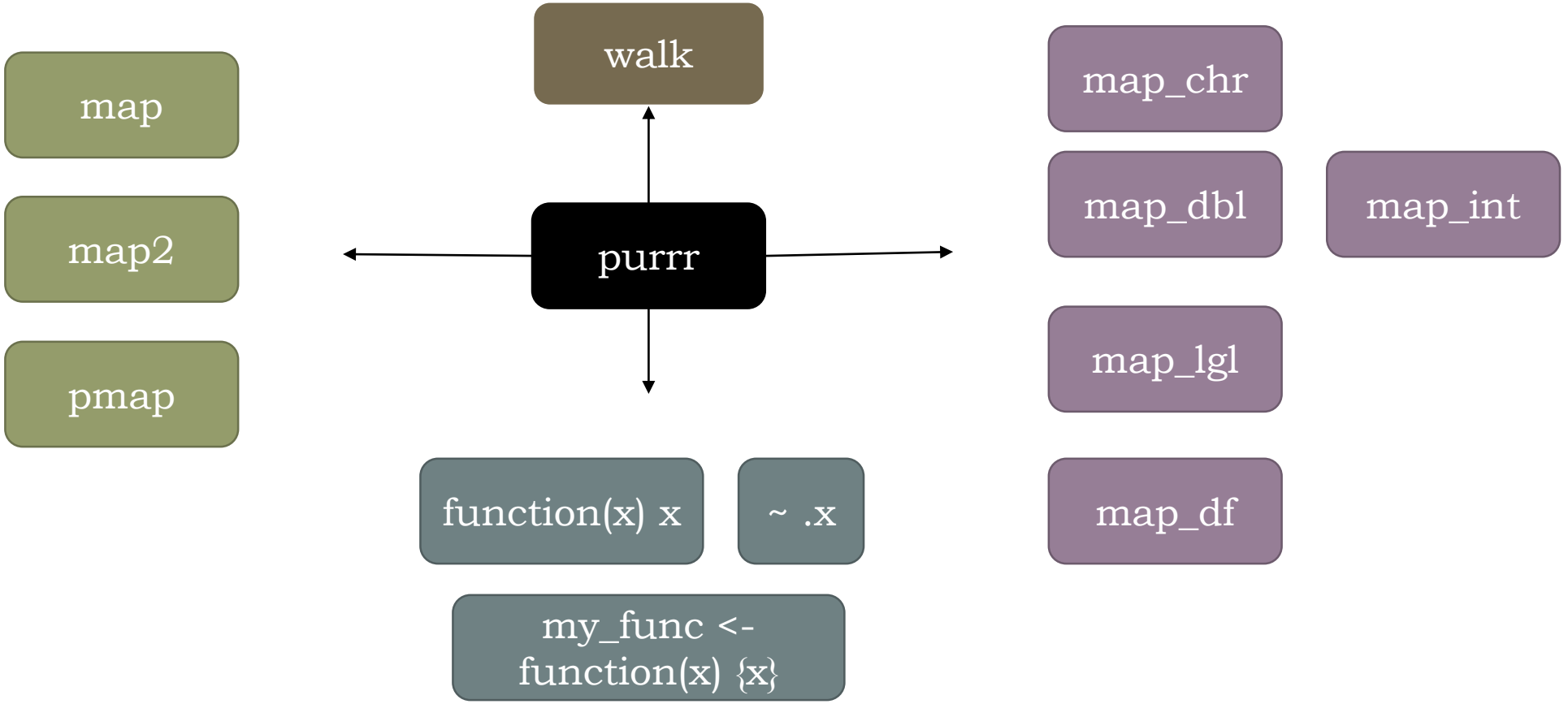
Using an inline function:

```
map_dbl(x, ~ mean(.x, na.rm = TRUE))
```

Map also passes the …, so you can just keep on specifying:

```
map_dbl(x, mean, na.rm = TRUE)
```

Functions in the purrr family



Exercise (question 1) (code available at 06-Purrr.R)

- Explain the differences between the following four code segments. Explain what happens in each:

```
iris %>%
  group_by(Species) %>%
  map_dbl(length)

iris %>%
  group_by(Species) %>%
  nest() %>%
  mutate(mean1 = map_int(data, length))

iris %>%
  group_by(Species) %>%
  nest() %>%
  mutate(mean2 = map_int(data, NROW))

iris %>%
  group_by(Species) %>%
  nest() %>%
  mutate(lm = map(data, function(df) lm(data = df, Sepal.Length ~ .)))
```

Exercise (question 2) (code available at 06-Purrr.R)

- Reading multiple files using map and extracting information out of them, using map_*

Exercise (question 3) (code available at 06-Purrr.R)

- Using walk to get the function's side effect.