Columbia University
In the City of New York

GR5206: lecture 3

Computational Statistics
And Introduction to Data Science

Thibault Vatter

Department of Statistics, Columbia University

9/20/2019

The two programming paradigms

- **Imperative:**
  - The programmer instructs the machine how to change its state.
  - Two kinds:
    - **Procedural:** groups instructions into procedures.
    - **Object-oriented:** groups instructions together with the part of the state they operate on.
- **Declarative:**
  - The programmer declares properties of the desired result, but not how to compute it.
  - Three kinds:
    - **Functional:** the output results of a series of function applications.
    - **Logic:** the output is the answer to a question about a system of facts and rules.
    - **Mathematical:** the output is the solution of an optimization problem.

What about R?

- A bit of everything:
  - Powerful but complex.
- Imperative:
  - Procedural: functions loaded with `source()`.
  - Object-oriented: the `S3` class system (and others).
- Declarative:
  - Mathematical: optimization with `optim` and specialized packages.
  - Functional: **the hearth** of R.

Functional programming languages

- What makes a programming language functional?
  - Many definitions but two common threads:
    - **First-class** functions.
    - **Pure** functions.
- **Functional style**:
  - Hard to describe exactly, but essentially:
    - Decompose a problem into small pieces, then solve each piece with a (combination of) function(s).
    - Each function is simple and straightforward to understand.
    - Complexity is handled by composing functions.

First-class functions

- Functions behave like any other data structure.
- In R, means that you can:
  - Assign them to variables.
  - Store them in lists.
  - Pass them as arguments to other functions.
  - Create them inside functions.
  - And even return them as the result of a function.

```r
f1 <- function(x) x
l1 <- list(
  mean,
  sd,
  median
)
y <- rnorm(1e1)
sapply(l1, function(f) f(y))
#> [1] -0.441  1.118 -0.246
```

Pure functions

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- Two main properties:
  - The output only depends on the inputs:
    - Call it again with the same inputs, get the same outputs.
    - Excludes functions like `runif()` or `read.csv()` (why?).
  - No side-effects:
    - E.g., no changing the value of a global variable, writing to disk, or displaying to the screen.
    - Excludes functions like `print()`, `write.csv()` and `<-`.
- Two remarks:
  - Much easier to reason about, but some downsides:
    - How to do data analysis without generate random numbers or read files from disk?
  - Strictly speaking:
    - R isn't a functional *language* (why?).
    - While you don't *have* to write pure functions, you often *should*.

Functional style

- Three techniques:
  - **Functionals**:
    - Replace many loops.
    - E.g., `lapply()`, `sapply()`.
    - The most important, used all the time in data analysis.
  - **Function factories**:
    - Functions that create functions.
    - Partition work between different parts of your code.
  - **Function operators**:
    - Functions that take/return functions as inputs/output.
    - Typically modify the operation of a function.

- Called **higher-order functions**

In \ Out	Vector	Function
Vector	Regular function	Function factory
Function	Functional	Function operator

Outline

Functionals

To become significantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. Messy code often hides bugs.
— *Bjarne Stroustrup*

- **Functional**:
  - Takes/returns a function/vector as an input/output.
  - `lapply()`, `apply()`, `tapply()`, purrr's `map()`, `integrate()` or `optim()`.

```
randomise <- function(f) f(runif(1e3))
randomise(mean)
#> [1] 0.509
randomise(mean)
#> [1] 0.498
randomise(sum)
#> [1] 492
```

Outline

- `purrr::map()`:
  - Combine multiple simple functionals to solve larger problems.
  - The 18 (!!) important variants of `purrr::map()`.
- `purrr::reduce()`.
- Predicates and the functionals using them.
- Some functionals in base R not members of those families.
- Focus on the purrr package:
  - Consistent interface that makes it easier to use/understand.
  - We'll compare to base R functions equivalents.

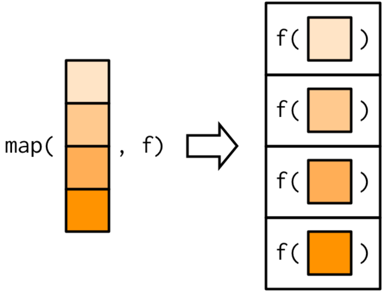
```r
library(purrr)
```

Outline

Warm-up: `purrr::map()`

- The most fundamental functional:
  - Takes a vector and a function.
  - Calls the function once for each element of the vector
  - Returns the results in a list.
- `map(1:3, f)` is equivalent to `list(f(1), f(2), f(3))`.
- The R base equivalent: `lapply()`.

```r
triple <- function(x) x * 3
map(1:3, triple)
#> [[1]]
#> [1] 3
#>
#> [[2]]
#> [1] 6
#>
#> [[3]]
#> [1] 9
```



How does that work?

- Simple implementation:
  - Allocate a list the same length as the input.
  - Fill in the list with a for loop.

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

- A few differences for the real implementation:
  - Written in C for performance.
  - Preserves names
  - Supports a few shortcuts.

Producing atomic vectors

- map() returns a list
- 4 more specific variants:
  - map_dbl(), map_chr(), map_int() and map_lgl().
- map_dbl() always returns a double vector.

```
map_dbl(mtcars, mean)
#>     mpg     cyl    disp      hp    drat      wt    qsec      vs
#>  20.091   6.188 230.722 146.688   3.597   3.217  17.849   0.438
#>      am    gear    carb
#>   0.406   3.688   2.812
```

- map_chr() always returns a character vector

```
map_chr(mtcars, typeof)
#>      mpg      cyl     disp       hp     drat       wt     qsec
#> "double" "double" "double" "double" "double" "double" "double"
#>       vs       am     gear     carb
#> "double" "double" "double" "double"
```

Producing atomic vectors con'd

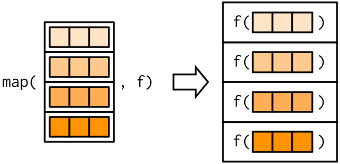
- `map_int()` always returns an integer vector.

```
map_int(mtcars, function(x) length(unique(x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

- `map_lgl()` always returns a logical vector.

```
map_lgl(mtcars, is.double)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

- Remarks:
  - Suffixes refer to the output.
  - But `map_*()` can take any type of vector as input.
- Examples rely on two facts:
  - `mtcars` is a data frame.
  - data frames are lists containing vectors of the same length.



Producing atomic vectors con'd

- Each call to the function must return a single value.

```
map_dbl(1:2, function(x) c(x, x))
#> Result 1 must be a single double, not an integer vector of length 2
```

- And obviously return the correct type.

```
map_dbl(1:2, as.character)
#> Error: Can't coerce element 1 from a character to a double
```

- In either case, use `map()` to see the problematic output!
- In base R:
  - `sapply()`.
    - Tries to simplify the result,.
    - Can return a list, a vector, or a matrix.
    - Difficult to program with, avoid in non-interactive settings.
  - `vapply()`.
    - `FUN.VALUE` to describe the output shape.
    - Verbosity: `vapply(x, mean, na.rm = TRUE, FUN.VALUE = double(1))` for `map_dbl(x, mean, na.rm = TRUE)`.

Anonymous functions and shortcuts

- map can use anonymous functions.

```
map_dbl(mtcars, function(x) length(unique(x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

- Less verbose shortcut.

```
map_dbl(mtcars, ~ length(unique(.x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

- Useful for generating random data.

```
x <- map(1:3, ~ runif(2))
str(x)
#> List of 3
#>  $ : num [1:2] 0.307 0.254
#>  $ : num [1:2] 0.656 0.839
#>  $ : num [1:2] 0.834 0.425
```

- Rule of thumb: a function spans lines/uses {}, give it a name.

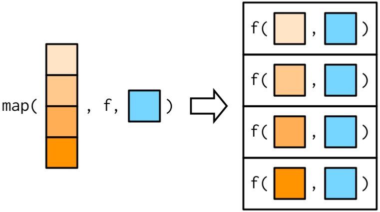
Extracting elements from a vector


```r
x <- list(
  list(-1, x = 1, y = c(2), z = "a"),
  list(-2, x = 4, y = c(5, 6), z = "b"),
  list(-3, x = 8, y = c(9, 10, 11))
)

# Select by name
map_dbl(x, "x")
#> [1] 1 4 8

# Or by position
map_dbl(x, 1)
#> [1] -1 -2 -3

# Or by both
map_dbl(x, list("y", 1))
#> [1] 2 5 9

# You'll get an error if a component doesn't exist:
map_chr(x, "z")
#> Result 3 must be a single string, not NULL of length 0
```

Passing arguments with ...

- To pass along additional arguments, use an anonymous function.

```
x <- list(1:5, c(1:10, NA))
map_dbl(x, ~ mean(.x, na.rm = TRUE))
#> [1] 3.0 5.5
```



- Or the simpler form.

```
map_dbl(x, mean, na.rm = TRUE)
#> [1] 3.0 5.5
```

- A subtle difference.

```
plus <- function(x, y) x + y
x <- c(0, 0, 0, 0)

map_dbl(x, plus, runif(1))
#> [1] 0.281 0.281 0.281 0.281

map_dbl(x, ~ plus(.x, runif(1)))
#> [1] 0.358 0.946 0.570 0.589
```

Map variants

- 23 primary variants of `map()`:
  - `map()`, `map_dbl()`, `map_chr()`, `map_int()`, `map_lgl()`
  - 18 (!!) more to learn.
  - Five new ideas:
    - Output same type as input with `modify()`
    - Iterate over two inputs with `map2()`.
    - Iterate with an index using `imap()`
    - Return nothing with `walk()`.
    - Iterate over any number of inputs with `pmap()`.

	List	Atomic	Same type	Nothing
One argument	`map()`	`map_lgl()`, ...	`modify()`	`walk()`
Two arguments	`map2()`	`map2_lgl()`, ...	`modify2()`	`walk2()`
One argument + index	`imap()`	`imap_lgl()`, ...	`imodify()`	`iwalk()`
N arguments	`pmap()`	`pmap_lgl()`, ...	—	`pwalk()`

Same type of output/input: `modify()`

```r
df <- data.frame(x = 1:3, y = 6:4)

map(df, ~ .x * 2)
#> $x
#> [1] 2 4 6
#>
#> $y
#> [1] 12 10  8

modify(df, ~ .x * 2)
#>   x  y
#> 1 2 12
#> 2 4 10
#> 3 6  8
```

- A simple implementation.

```r
simple_modify <- function(x, f, ...) {
  for (i in seq_along(x)) {
    x[[i]] <- f(x[[i]], ...)
  }
  x
}
```

Two inputs: map2() and friends

Columbia University in the City of New York

- How do we find the vector of weighted means?

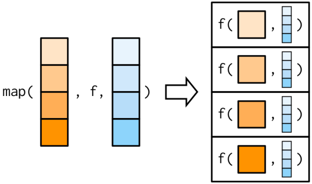
```
xs <- map(1:8, ~ runif(10))
xs[[1]][[1]] <- NA
ws <- map(1:8, ~ rpois(10, 5) + 1)
```

- Use map_dbl() to compute the unweighted means.

```
map_dbl(xs, mean)
#> [1]    NA 0.520 0.510 0.634 0.475 0.560 0.486 0.495
```

- Passing ws as an additional argument doesn't work.

```
map_dbl(xs, weighted.mean, w = ws)
#> Error in weighted.mean.default(.x[[i]], ...): 'x' and 'w' must have the same
```



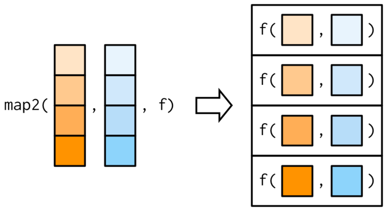
map2() and friends cont'd

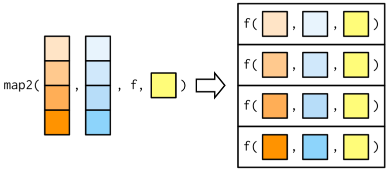
- Both arguments are varied in each call.

```
map2_dbl(xs, ws, weighted.mean)
#> [1]    NA 0.467 0.470 0.638 0.516 0.550 0.471 0.524
```

- Additional arguments still go afterwards.

```
map2_dbl(xs, ws, weighted.mean, na.rm = TRUE)
#> [1] 0.458 0.467 0.470 0.638 0.516 0.550 0.471 0.524
```





No outputs: `walk()` and friends


```r
welcome <- function(x) {
  cat("Welcome ", x, "!\n", sep = "")
}
names <- c("Hadley", "Jenny")

# As well as generate the welcomes, it also shows
# the return value of cat()
map(names, welcome)
#> Welcome Hadley!
#> Welcome Jenny!
#> [[1]]
#> NULL
#>
#> [[2]]
#> NULL

## Avoid this with walk
walk(names, welcome)
#> Welcome Hadley!
#> Welcome Jenny!
```

Iterating over values and indices

Columbia University in the City of New York

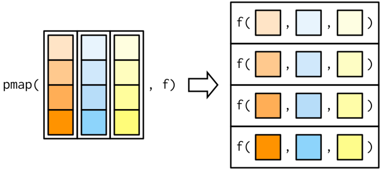
- Three basic ways to loop over a vector with `for`:
  - Over the elements: `for (x in xs) f(xs)`
  - Over the names: `for (nm in names(xs)) f(nm)`
  - Over the indices: `for (i in seq_along(xs)) f(i)`
- First kind: similar to `map(xs, f)`.
- The other two: `imap(xs, f)`.
  - Same as `map2(xs, names(xs), f)` if `xs` as names.
  - Same as `map2(xs, seq_along(xs), f)` otherwise.

```
imap_chr(iris, ~ paste0("The first value of ", .y, " is ", .x[[1]]))
#>                             Sepal.Length
#> "The first value of Sepal.Length is 5.1"
#>                              Sepal.Width
#>  "The first value of Sepal.Width is 3.5"
#>                             Petal.Length
#> "The first value of Petal.Length is 1.4"
#>                              Petal.Width
#>  "The first value of Petal.Width is 0.2"
#>                                  Species
#>   "The first value of Species is setosa"
```

Any number of inputs: pmap()

- map() and map2()... map3(), map4(), map5()?
- Instead, there is pmap():
  - Supply it a single list, which contains any number of arguments.
  - In most cases, a list of equal-length vectors (e.g., a data frame).

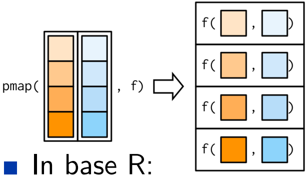
```
params <- tibble::tribble(
  ~ n, ~ min, ~ max,
   1L,     0,     1,
   2L,    10,   100,
   3L,   100,  1000
)

pmap(params, runif)
#> [[1]]
#> [1] 0.15
#>
#> [[2]]
#> [1] 17.3 74.8
#>
#> [[3]]
#> [1] 933 994 402
```

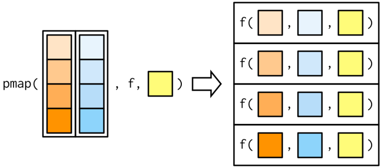


pmap() cont'd

- `pmap(list(x, y), f, na.rm = TRUE)` is the same as `map2(x, y, f, na.rm = TRUE)`.

```
pmap_dbl(list(xs, ws), weighted.mean)
#> [1]    NA 0.467 0.470 0.638 0.516 0.550 0.471 0.524

pmap_dbl(list(xs, ws), weighted.mean, na.rm = TRUE)
#> [1] 0.458 0.467 0.470 0.638 0.516 0.550 0.471 0.524
```



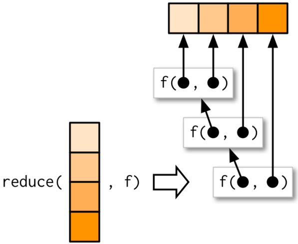


- In base R:
  - `Map()`:
    - Vectorises over all arguments
    - Cannot supply arguments that do not vary.
  - `mapply()`:
    - Multidimensional version of `sapply()`.

Outline

Reduce family

- The next most important (family of) functionals.
  - Much smaller (two main variants).
  - Powers the map-reduce framework.
- `purrr::reduce()`:
  - Takes a vector of length n.
  - Produces a vector of length 1 by calling a function with a pair of values at a time.
  - `reduce(1:4, f)` is equivalent to `f(f(f(1, 2), 3), 4)`.



Reduce family cont'd

- Useful to generalize a function that works with two inputs to work with any number of inputs.
- Problem: find the values that occur in every element.

```
l <- map(1:4, ~ sample(1:10, 15, replace = T))
str(l)
#> List of 4
#>  $ : int [1:15] 7 5 7 2 9 5 2 8 10 5 ...
#>  $ : int [1:15] 9 2 1 4 7 1 3 5 5 4 ...
#>  $ : int [1:15] 8 7 5 1 6 6 10 9 2 1 ...
#>  $ : int [1:15] 10 3 6 2 7 5 6 9 8 4 ...
```

- Two solutions:

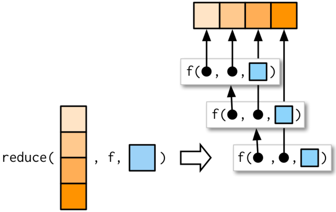
```
out <- l[[1]]
out <- intersect(out, l[[2]])
out <- intersect(out, l[[3]])
out <- intersect(out, l[[4]])
out
#> [1]  7  5  2  9 10  4  3
```

```
reduce(l, intersect)
#> [1]  7  5  2  9 10  4  3
```

Reduce family cont'd

- Can also pass additional arguments.
- Simple implementation.

```
simple_reduce <- function(x, f, ...) {
  out <- x[[1]]
  for (i in seq(2, length(x))) {
    out <- f(out, x[[i]], ...)
  }
  out
}
```



- In base R:
  - Reduce().
    - The function comes first, followed by the vector.
    - No way to supply additional arguments.

Accumulate


```
accumulate(l, intersect)
#> [[1]]
#>  [1]  7  5  7  2  9  5  2  8 10  5  4  2  3  6  5
#>
#> [[2]]
#> [1]  7  5  2  9 10  4  3
#>
#> [[3]]
#> [1]  7  5  2  9 10  4  3
#>
#> [[4]]
#> [1]  7  5  2  9 10  4  3

x <- c(4, 3, 10)
reduce(x, `+`)
#> [1] 17
reduce(x, `+`) == sum(x)
#> [1] TRUE
accumulate(x, `+`)
#> [1]  4  7 17
accumulate(x, `+`) == cumsum(x)
#> [1] TRUE TRUE TRUE
```

Outline

Predicate functionals

- A **predicate**:
  - Function that returns a single `TRUE` or `FALSE`.
  - E.g., `is.character()`, `is.null()`, or `all()`.
  - A predicate **matches** a vector if it returns `TRUE`.
- A **predicate functional**:
  - Applies a predicate to each element of a vector.
  - 6 functions in 3 pairs.
  - `some(.x, .p)`/`every(.x, .p)`.
    - Returns `TRUE` if *any/all* element matches.
    - Similar to `any(map_lgl(.x, .p))`/`all(map_lgl(.x, .p))`.
    - But terminate early.
  - `detect(.x, .p)`/`detect_index(.x, .p)`.
    - Returns the *value/location* of the first match.
  - `keep(.x, .p)`/`discard(.x, .p)`.
    - *Keeps/drops* all matching elements.

Predicate functionals cont'd


```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
detect(df, is.factor)
#> [1] a b c
#> Levels: a b c
detect_index(df, is.factor)
#> [1] 2

str(keep(df, is.factor))
#> 'data.frame':    3 obs. of  1 variable:
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
str(discard(df, is.factor))
#> 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
```

Map variants

```r
df <- data.frame(
  num1 = c(0, 10, 20),
  num2 = c(5, 6, 7),
  chr1 = c("a", "b", "c"),
  stringsAsFactors = FALSE
)

str(map_if(df, is.numeric, mean))
#> List of 3
#>  $ num1: num 10
#>  $ num2: num 6
#>  $ chr1: chr [1:3] "a" "b" "c"
str(modify_if(df, is.numeric, mean))
#> 'data.frame':    3 obs. of  3 variables:
#>  $ num1: num  10 10 10
#>  $ num2: num  6 6 6
#>  $ chr1: chr  "a" "b" "c"
str(map(keep(df, is.numeric), mean))
#> List of 2
#>  $ num1: num 10
#>  $ num2: num 6
```

Outline

Base functionals

- Some base R functionals have no `purrr` equivalent:
  - Working with two-dimensional and higher vectors:
    - `base::apply()`: summarizes by collapsing rows/columns to a single value.
  - Mathematical tools:
    - `integrate()`: the area under the curve defined by `f()`
    - `uniroot()`: where `f()` hits zero
    - `optimise()`: the location of the lowest (or highest) value of `f()`

Matrices and arrays: `base::apply()`

- Summarizes by collapsing rows/columns to a single value.

```
a2d <- matrix(1:20, nrow = 5)
apply(a2d, 1, mean)
#> [1]  8.5  9.5 10.5 11.5 12.5
apply(a2d, 2, mean)
#> [1]  3  8 13 18
```

- Two caveats:
  - No control over the output type.
  - Doesn't work (well) with data frames.

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
apply(df, 2, mean)
#> Warning in mean.default(newX[, i], ...): argument is not numeric or
#> logical: returning NA

#> Warning in mean.default(newX[, i], ...): argument is not numeric or
#> logical: returning NA
#>  x  y
#> NA NA
```

Mathematical tools

```r
integrate(sin, 0, pi)
#> 2 with absolute error < 2.2e-14
str(uniroot(sin, pi * c(1 / 2, 3 / 2)))
#> List of 5
#>  $ root      : num 3.14
#>  $ f.root    : num 1.22e-16
#>  $ iter      : int 2
#>  $ init.it   : int NA
#>  $ estim.prec: num 6.1e-05
str(optimise(sin, c(0, 2 * pi)))
#> List of 2
#>  $ minimum  : num 4.71
#>  $ objective: num -1
str(optimise(sin, c(0, pi), maximum = TRUE))
#> List of 2
#>  $ maximum  : num 1.57
#>  $ objective: num 1
```

Outline

Function factories

- A **function factory** is a function that makes functions.
- Problem: replace all the missing values with `NA`s.

```
set.seed(123)
(df <- data.frame(replicate(4, sample(c(1:10, -99), 4, rep = TRUE))))
#>   X1  X2  X3  X4
#> 1  3   6   6   5
#> 2  3 -99   9   3
#> 3 10   5  10 -99
#> 4  2   4 -99   9
```

- A first approach.

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
modify(df, fix_missing)
#>   X1 X2 X3 X4
#> 1  3  6  6  5
#> 2  3 NA  9  3
#> 3 10  5 10 NA
#> 4  2  4 NA  9
```

Function factories cont'd

- What about this dataset?

```
df[1:2, 1] <- -999
df
#>     X1  X2  X3  X4
#> 1 -999   6   6   5
#> 2 -999 -99   9   3
#> 3   10   5  10 -99
#> 4    2   4 -99   9
```

- Naive approach.

```
fix_missing_99 <- function(x) {
  x[x == -99] <- NA
  x
}
fix_missing_999 <- function(x) {
  x[x == -999] <- NA
  x
}
```

- Or using a function factory.

```
missing_fixer <- function(na_value) {
  function(x) {
    x[x == na_value] <- NA
    x
  }
}
fix_missing_99 <- missing_fixer(-99)
fix_missing_999 <- missing_fixer(-999)
```

- **More compelling uses for the concept in 2 weeks!**

A caveat

- Another example.

```r
power1 <- function(exp) {
  function(x) {
    x ^ exp
  }
}

square <- power1(2)
square(3)
#> [1] 9

cube <- power1(3)
cube(3)
#> [1] 27
```

- What will the following code return?

```r
x <- 2
square <- power1(x)
x <- 3
square(2)
```

The solution: `force()`

- Remember that R is **lazy**.

```r
power1 <- function(exp) {
  function(x) {
    x ^ exp
  }
}


x <- 2
square <- power1(x)
x <- 3
square(2)
#> [1] 8
```

```r
power2 <- function(exp) {
  force(exp)
  function(x) {
    x ^ exp
  }
}

x <- 2
square <- power2(x)
x <- 3
square(2)
#> [1] 4
```

- Don't forget to force the evaluation with `force()`!

Outline

Function operators

- Functions that takes one (or more) functions as input and returns a function as output.

```r
chatty <- function(f) {
  function(x, ...) {
    cat("Processing ", x, "\n", sep = "")
    f(x, ...)
  }
}

f <- function(x) x ^ 2
map_dbl(c(3, 2, 1), chatty(f))
#> Processing 3
#> Processing 2
#> Processing 1
#> [1] 9 4 1
```

- Closely related to function factories
  - They're just a function factory that takes a function as input.
  - Nothing you can't do without, powerful to factor out complexity.
- Typically paired with functionals.
- For Python users: decorators is just another name!

Dealing with failure using `safely()`

- The modified function always returns a list with two elements:
  1. `result` is the original result.
  2. `error` is an error object.

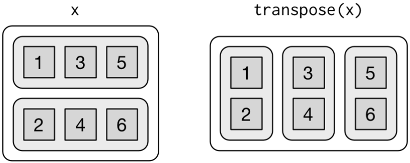
```
safe_log <- safely(log)
str(safe_log(10))
#> List of 2
#>  $ result: num 2.3
#>  $ error : NULL
str(safe_log("a"))
#> List of 2
#>  $ result: NULL
#>  $ error :List of 2
#>   ..$ message: chr "non-numeric argument to mathematical function"
#>   ..$ call   : language .Primitive("log")(x, base)
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

safely() and map()


```
x <- list(1, 10, "a")
y <- map(x, safely(log))
str(y)
#> List of 3
#>  $ :List of 2
#>   ..$ result: num 0
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 2.3
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: NULL
#>   ..$ error :List of 2
#>   .. ..$ message: chr "non-numeric argument to mathematical function"
#>   .. ..$ call   : language .Primitive("log")(x, base)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condit"..
```

transpose()


```r
y <- transpose(y)
str(y)
#> List of 2
#>  $ result:List of 3
#>   ..$ : num 0
#>   ..$ : num 2.3
#>   ..$ : NULL
#>  $ error :List of 3
#>   ..$ : NULL
#>   ..$ : NULL
#>   ..$ :List of 2
#>   .. ..$ message: chr "non-numeric argument to mathematical function"
#>   .. ..$ call   : language .Primitive("log")(x, base)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condit"..
```



Typical use


```r
is_ok <- map_lgl(y$error, is_null)
x[!is_ok]
#> [[1]]
#> [1] "a"
flatten_dbl(y$result[is_ok])
#> [1] 0.0 2.3
```

Two other useful adverbs

- `possibly()`: "simpler" than `safely()`, because you give it a default value to return when there is an error.

```
map_dbl(x, possibly(log, NA_real_))
#> [1] 0.0 2.3  NA
```

- `quietly()`: instead of capturing errors, it captures printed output, messages, and warnings.

```
map(list(1, -1), quietly(log)) %>% str()
#> List of 2
#>  $ :List of 4
#>   ..$ result  : num 0
#>   ..$ output  : chr ""
#>   ..$ warnings: chr(0)
#>   ..$ messages: chr(0)
#>  $ :List of 4
#>   ..$ result  : num NaN
#>   ..$ output  : chr ""
#>   ..$ warnings: chr "NaNs produced"
#>   ..$ messages: chr(0)
```

Caching computations with `memoise()`

- Provided by the `memoise` package.
- **Memoises** a function
  - The function remembers previous inputs/returns cached results.
  - Classic CS tradeoff of memory versus speed:
    - A memoised function is faster, but uses more memory.

```r
slow_fct <- function(x) {
  Sys.sleep(1)
  x * 10 * runif(1)
}

system.time(print(slow_fct(1)))
#> [1] 6.93
#>    user  system elapsed
#>   0.001   0.000   1.001
system.time(print(slow_fct(1)))
#> [1] 6.41
#>    user  system elapsed
#>   0.005   0.000   1.007
```

```r
library(memoise)
fast_fct <- memoise(slow_fct)


system.time(print(fast_fct(1)))
#> [1] 9.94
#>    user  system elapsed
#>   0.001   0.000   1.002
system.time(print(fast_fct(1)))
#> [1] 9.94
#>    user  system elapsed
#>   0.043   0.000   0.044
```

Fibonacci series

- Defined recursively:
  - $f(0) = 0$, $f(1) = 1$,
  - And then $f(n) = f(n-1) + f(n-2)$.

```r
fib <- function(n) {
  if (n < 2) return(1)
  fib(n - 2) + fib(n - 1)
}

system.time(fib(23))
#>    user  system elapsed
#>   0.089   0.020   0.109
system.time(fib(24))
#>    user  system elapsed
#>   0.159   0.000   0.159
```

```r
fib2 <- memoise(function(n) {
  if (n < 2) return(1)
  fib2(n - 2) + fib2(n - 1)
})

system.time(fib2(23))
#>    user  system elapsed
#>   0.073   0.000   0.073
system.time(fib2(24))
#>    user  system elapsed
#>   0.002   0.000   0.002
```

- An example of **dynamic programming**:
  - Complex problem broken down into overlapping subproblems.
  - Remembering the results of a subproblem considerably improves performance.