Data Science for Business Analytics

Control Flow and Functions

HEC Lausanne
Professor Alex [aleksandr.shemendyuk@unil.ch]

Today

- **Control flow**:
  - Allows to execute code depending on conditions
    - `if` statements
    - `ifelse` statements
  - Allows to run code repeatedly
    - `for` loops
    - `while` loops
- **Functions**:
  - Identify repeated patterns in code
  - Encapsulate code into reusable functions
- **Iterations**:
  - Apply functions to multiple elements
  - `lapply`, `sapply`, `vapply`, `mapply`, `apply`
  - `purrr` package with `map`, `reduce`, `walk`, `pmap`

Agenda

`if()` statements

The basic idea of `if` statements: if a `condition` is

- `TRUE`, then execute `true_action`
- `FALSE`, then execute an optional `false_action`.

```
if (condition) true_action
if (condition) true_action else false_action
```

Typically, actions are compound statements contained within `{`.

```
grade <- function(x) {
  if (x > 90) {
    "A"
  } else if (x > 80) {
    "B"
  } else if (x > 50) {
    "C"
  } else {
    "F"
  }
}
```

if() statements cont'd

- if may return a value, so you can assign it to a variable:[1]

```r
x1 <- if (TRUE) 1 else 2
x2 <- if (FALSE) 1 else 2

c(x1, x2)
#> [1] 1 2
```

- When using if without else:
  - Returns NULL if the condition is FALSE.
  - Useful with functions like c()/paste() dropping NULL inputs.

```r
greet <- function(name, birthday = FALSE) {
  paste0("Hi ", name, if (birthday) " and HAPPY BIRTHDAY")  # (2)
}
greet("Maria", FALSE)
greet("Jaime", TRUE)
#> [1] "Hi Maria"
#> [1] "Hi Jaime and HAPPY BIRTHDAY"
```

2. paste0() concatenates strings without spaces.

[1] Only do this when it fits on one line; otherwise it's hard to read.

Invalid `if` inputs

- `if` expects a single logical value.
- So, the condition should evaluate to a single `TRUE` or `FALSE` value.

```
if ("x") 1
#> Error in if ("x") 1: argument is not interpretable as logical

if (logical()) 1
#> Error in if (logical()) 1: argument is of length zero

if (NA) 1
#> Error in if (NA) 1: missing value where TRUE/FALSE needed

if (c(TRUE, FALSE)) 1
#> Error in if (c(TRUE, FALSE)) 1: the condition has length > 1
```

Vectorised `if()` statements

- `ifelse()` is a vectorised version of `if`.
  - **Structure**: `ifelse(condition, true_action, false_action)`.[2]
  - **Output**: A vector of the same length as the condition.

```
x <- 1:9
ifelse(x %% 5 == 0, "XXX", as.character(x))
#> [1] "1"   "2"   "3"   "4"   "XXX" "6"   "7"   "8"   "9"
ifelse(x %% 2 == 0, "even", "odd")
#> [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"
```

- For any number of condition-vector pairs use `dplyr::case_when()`.

```
dplyr::case_when(
  x %% 6 == 0 ~ "fizz buzz",
  x %% 3 == 0 ~ "fizz",
  x %% 2 == 0 ~ "buzz",
  .default = as.character(x)
)
#> [1] "1"         "buzz"      "fizz"      "buzz"      "5"
#> [6] "fizz buzz" "7"         "buzz"      "fizz"
```

[2] Use `true_action` and `false_action` of the same <type>.

Agenda

`for` loops

- `for` loops are used to iterate over a sequence of elements.
- **Structure**: `for (element in sequence) { code }`.
- Allows to repeat the same operation for each `element` in the `sequence`.

```r
for (i in 1:3) {
  print(i)
}
```

`for` loops (cont'd)

Bad coding style:

```
set.seed(123)
tbl <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

output <- c(
  median(tbl$a), median(tbl$b),
  median(tbl$c), median(tbl$d)
)
print(output)
#> [1] -0.0798  0.3803 -0.6770  0.4902
```

Better coding style:

```
set.seed(123)
tbl <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

output <- numeric(length(tbl))
for (i in seq_along(tbl)) {
  output[i] <- median(tbl[[i]])
}
print(output)
#> [1] -0.0798  0.3803 -0.6770  0.4902
```

1. `rnorm()`: random numbers from a standard normal distribution.
2. `seq_along(tbl)`: returns a sequence of integers from 1 to the number of columns in `tbl`.

`for` Early Exit

- There are situations where you want to **exit a loop early**.
  - `next`: skip the current iteration and continue with the next one.
  - `break`: exit a loop prematurely.

```
for (i in 1:10) {
  if (i < 3)
    next

  print(i)

  if (i >= 5)
    break
}
#> [1] 3
#> [1] 4
#> [1] 5
```

`for` Common Pitfalls

- **Pitfall 1**: Modifying the loop variable inside the loop.

```
x <- 1:5
for (i in x) {
  x <- x[1 : length(x)-1]
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
```

- At the end of the first iteration, `x` is `1:4`. The code compiler preallocates the loop variable `i` to `1:5`, so it will iterate over `1:5` even though `x` is now `1:4`.

- **Pitfall 2**: Using `for` loops to grow objects.

```
output <- numeric(0)
for (i in 1:5) {
  output <- c(output, i)
}
```

- This is inefficient because the object `output` is internally copied at each iteration.

- **Pitfall 3**: Using `for` loops when vectorised operations are possible.

Related tools

- `for` loops:
  - Useful when known in advance the set of values to iterate over.
  - Otherwise, use `while` loops:
    - **Structure**: `while (condition) { code }`.
    - Performs `code` while `condition` is `TRUE`.
    - Possible to write any `for` using `while`.
    - Good practice is to prefer `for` loops over `while`.
- Generally speaking you **shouldn't need** to use loops for **data analysis tasks**. We'll see better solutions.

Agenda

Function fundamentals

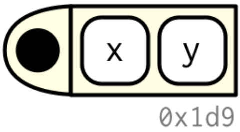
- Two **important ideas**:
  - Functions can be broken down into three components: *arguments*, *body*, and *environment*.
  - Functions are **objects**, just as vectors are objects.

- **The basics:**
  - How to create functions.
  - The three main components of a function.
  - How can a function exit.
  - Anonymous functions.
  - The special ... argument.
- **Lexical scoping:** how R finds the value associated with a given name.
  - Name masking.
  - Functions versus variables.
  - A fresh start.
  - Dynamic lookup.
- **The special ... argument:** how to pass on extra arguments to another function.
- **Exiting a function:** and exit handlers.
- **Function forms:** the prefix form and more.

Function components

- A function has three parts:
  - `formals()`: function arguments.
  - `body()`: the code inside the function.
  - `environment()`: the data structure determining how the function finds the values associated with the names.

```
f02 <- function(x, y) {
  x + y
}
formals(f02)
#> $x
#>
#>
#> $y
body(f02)
#> {
#>     x + y
#> }
environment(f02)
#> <environment: R_GlobalEnv>
```



Primitive functions

- One exception to the three components rule.
- Call C code directly.

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")
`[`
#> .Primitive("[")
```

- The `<type>` is either `builtin` or `special`.

```
typeof(sum)
#> [1] "builtin"
typeof(`[`)
#> [1] "special"
```

- `formals()`, `body()`, and `environment()` are all `NULL`.

```
formals(sum)
#> NULL
body(sum)
#> NULL
environment(sum)
#> NULL
```

Exiting a function

- Most functions exit in one of two ways:
  - `return` a value, indicating success.
  - Throw an `error`, indicating failure.
- In the next few slides:
  - **Return values**.
    - Implicit versus explicit.
    - Visible versus invisible.
  - **Errors**.

Implicit versus explicit returns

- **Implicit**: the last evaluated expression is the return value.

```r
j01 <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
j01(5)
#> [1] 0
j01(15)
#> [1] 10
```

- **Explicit**: uses `return()` to return a value.

```r
j02 <- function(x) {
  if (x < 10) {
    return(0)
  } else {
    return(10)
  }
}
```

Errors

- If a function cannot complete its assigned task, it should throw an error using `stop()`:
  - Immediately terminates the execution of the function.
  - Indicates that something has gone wrong, and forces the user to deal with the problem.

```
j05 <- function() {
  stop("I'm an error")
  return(10)
}
j05()
#> Error in j05(): I'm an error
```

- Some languages rely on special return values to indicate problems, but in R you should always throw an error.

Anonymous function

- Use of standard functions:
  - Create a function object using `function`.
  - Bind it to a name with using `<-`.

```
f01 <- function(x) {
  sin(x)
}
```

- … but the binding step is not compulsory!
- A function without a name is called an **anonymous function**:

```
integrate(function(x) sin(x), 0, pi)
#> 2 with absolute error < 2.2e-14

sapply(1:10, function(x) x + 1)
#>  [1]  2  3  4  5  6  7  8  9 10 11
```

... (dot-dot-dot)

- The special argument `...`
  - Makes a function take any number of **additional arguments**.
  - In other programming languages they are often called *varargs* (short for variable arguments).
- Additional arguments can be passed to another function.

```
i01 <- function(y, z) {
  list(y = y, z = z)
}

i02 <- function(x, ...) {
  i01(...)
}

str(i02(x = 1, y = 2, z = 3))
#> List of 2
#>  $ y: num 2
#>  $ z: num 3
```

Agenda

Lexical scoping

- **Lexical scoping:** the most common scoping rule in programming languages.
  - Determines where to look up the values of names.
  - Based on how a function is defined, not how it is called.

```
x <- 10
g01 <- function() {
  x <- 20
  return(x)
}

g01()
#> [1] 20
```

- R uses follows four primary rules:
  - Name masking
  - Functions versus variables
  - A fresh start
  - Dynamic lookup

Name masking

- Names **defined inside** a function **mask names** defined outside.

```
x <- 10
y <- 20
g02 <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
g02()
#> [1] 1 2
```

- If a name isn't defined inside a function, R looks one level up.

```
x <- 2
y <- 20
g03 <- function() {
  y <- 1
  c(x, y)
}
g03()
#> [1] 2 1
y
#> [1] 20
```

Name masking cont'd

- Same applies if a function is defined inside another function:
    1. First, R looks inside the current function.
    2. Then, where that function was defined,
    3. and so on, all the way up to the global environment.
    4. Finally, in other loaded packages.

```r
x <- 1
g04 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
g04()
#> [1] 1 2 3
```

Functions versus variables

- Functions are objects, so the same scoping rules apply.

```
g07 <- function(x) x + 1
g08 <- function() {
  g07 <- function(x) x + 100
  g07(10)
}
g08()
#> [1] 110
```

- When a **function** and a **non-function** share the **same name**, the rules get a little more complicated.[3] For function calls, R ignores non-functions when scoping.

```
g09 <- function(x) x + 100
g10 <- function() {
  g09 <- 10
  g09(g09)
}
g10()
#> [1] 110
```

[3]Using the *same name* for different things **should be avoided!** For example, create your list as `my_list <- list(...)` and don't do `list <- list(...)`.

A fresh start

- What happens to values between invocations of a function?
- What happens the first time you run g11() function?
- What happens the second time?

```
g11 <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  return(a)
}

g11()
#> [1] 1

g11()
#> [1] 1
```

Dynamic lookup

- The output of a function can depend on objects outside of its environment, because:
  - Lexical scoping determines where, not when, to look for values.
  - R looks for values when the function is ran, not when the function is created.

```r
g12 <- function() x + 1
x <- 15
g12()
#> [1] 16

x <- 20
g12()
#> [1] 21
```

- Can be quite annoying.
  - With spelling mistakes, no error when creating a function.
  - Depending on the global environment, maybe not even an error when running the function.

Agenda

For loops vs. functionals

- Generate a `tibble`:

```
set.seed(123)
tbl <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

- Mean of every column using `for` loop:

```
output <- vector("double", length(tbl))
for (i in seq_along(tbl)) {
  output[[i]] <- mean(tbl[[i]])
}

output
#> [1]  0.0746  0.2086 -0.4246  0.3220
```

- Mean of every column using a custom function `col_mean`:

```
col_mean <- function(tb) {
  output <- vector("double", length(tb))
  for (i in seq_along(tb)) {
    output[i] <- mean(tb[[i]])
  }
  return(output)
}

col_mean(tbl)
#> [1]  0.0746  0.2086 -0.4246  0.3220
```

How about other quantities?

```
col_median <- function(tb) {
  output <- vector("double", length(tb))
  for (i in seq_along(tb)) {
    output[i] <- median(tb[[i]])
  }
  return(output)
}

col_sd <- function(tb) {
  output <- vector("double", length(tb))
  for (i in seq_along(tb)) {
    output[i] <- sd(tb[[i]])
  }
  return(output)
}

col_median(tbl)
#> [1] -0.0798  0.3803 -0.6770  0.4902
col_sd(tbl)
#> [1] 0.954 1.038 0.931 0.527
```

- What's "wrong" here? Too much code duplication!

A simple “functional”

```r
col_summary <- function(tb, fun) {
  output <- vector("double", length(tb))
  for (i in seq_along(tb)) {
    output[i] <- fun(tb[[i]])
  }
  return(output)
}

col_summary(tbl, median)
#> [1] -0.0798  0.3803 -0.6770  0.4902
col_summary(tbl, mean)
#> [1]  0.0746  0.2086 -0.4246  0.3220
```

The two programming paradigms

- **Imperative:**
  - The programmer instructs the machine how to change its state.
  - Examples:
    - **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.
  - Examples:
    - **Functional:** the output results of a series of function applications.
    - **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

- Functional programming (FP):
  - Uses **functions that return functions** as output.
  - Passes functions as arguments to others function.
  - Much more in the Advanced-R book chapter on FP
- 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, it means that you can:
  - **Assign** them to variables.
  - **Store** them in lists.
  - **Pass** them as arguments to other functions.
  - **Create** them inside functions.
  - **Return** them as the result of a function.

```r
function_list <- list(
  avg = mean,
  std = sd,
  med = median,
  max = function(x) max(x)
)

y <- rnorm(1e2) # 1*10^2
sapply(function_list, function(f) f(y))
#>      avg      std      med      max
#> -0.00721  0.93268 -0.05029  2.18733
```

Pure functions

- 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 `<-`.
- Some downsides:
  - How to do data analysis without generating random numbers or reading files from a disk?
  - While you don't *have* to write pure functions, you often *should*.

Functional style

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

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

Agenda

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::map()`, `integrate()` or `optim()`.

```r
randomise <- function(f) f(runif(1e3))
randomise(mean)
#> [1] 0.491
randomise(mean)
#> [1] 0.501
randomise(sum)
#> [1] 503
```

Outline

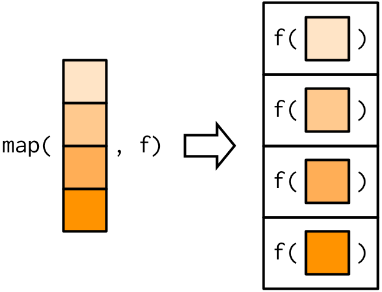
- `purrr::map()`:
  - The basic map functions
    1. Take a vector as input.
    2. Apply a function to each element.
    3. Return a new vector that's the same length as the input.
  - The return type is determined by the *suffix*:
    - `map()` returns a list.
    - `map_lgl()` returns a logical vector.
    - `map_int()` returns an integer vector.
    - `map_dbl()` returns a double vector.
    - `map_chr()` returns a character vector.
- `purrr::reduce()`.
- Predicates and the functionals using them.
- Mathematical functionals.
- Focus on the `purrr` package:

```
library(purrr)
```

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

- The most fundamental functional:
    1. Takes a vector and a function as inputs.
    2. Calls the function once for each element of the vector.
    3. 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 this work?

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

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

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

Producing atomic vectors

- `map()` returns a list.
- 4 more specific variants are available:
  - `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 (cont'd)

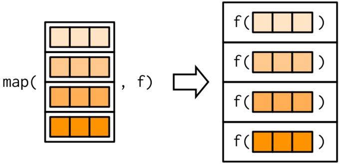
- `map_int()` always returns an integer vector.

```
map_int(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
```

- `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 (cont'd)

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

```
map_dbl(1:2, function(x) c(x, x))
#> Error in `map_dbl()`:
#> i In index: 1.
#> Caused by error:
#> ! Result must be length 1, not 2.
```

- And return the **correct type**.

```
map_dbl(1:2, as.character)
#> Error in `map_dbl()`:
#> i In index: 1.
#> Caused by error:
#> ! Can't coerce from a string to a double.
```

- In either case, use `map()` to see the problematic output!

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] 0.11 0.146
#>  $ : num [1:2] 0.317 0.607
#>  $ : num [1:2] 0.947 0.423
```

- If a function spans lines or uses {body}, give it a name.

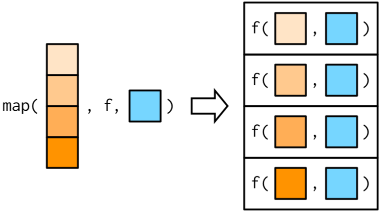
Passing arguments with ...

- To pass 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 in a simpler form:

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



- A subtle difference in the two approaches:

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

map_dbl(x, plus, runif(1))
#> [1] 0.926 0.926 0.926 0.926

map_dbl(x, ~ plus(.x, runif(1)))
#> [1] 0.0996 0.8663 0.2605 0.2840
```

Map variants

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

	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()`

Two inputs: map2() and friends

- How do we find the vector of weighted means?

```
xs <- map(1:4, ~ runif(10))
xs <- xs %>% purrr::assign_in(c(1, 2), NA) # xs[[1]][[2]] <- NA
ws <- map(1:4, ~ rpois(10, 5) + 1)
```

- Use `map_dbl()` to compute the unweighted means.

```
map_dbl(xs, mean)
#> [1]    NA 0.487 0.495 0.439
```

- Passing `ws` as an additional argument doesn't work because it's a `list`.

```
map_dbl(xs, weighted.mean, w = ws)
#> Error in `map_dbl()`:
#> i In index: 1.
#> Caused by error in `weighted.mean.default()`:
#> ! 'x' and 'w' must have the same length
```

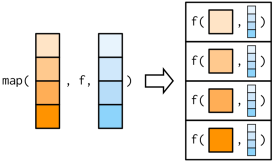


Figure 1: Here, in our example each square block is a numeric vector.

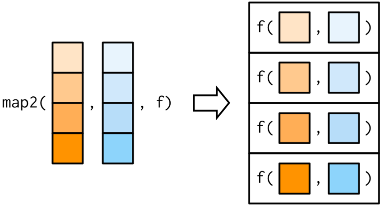
Two inputs: `map2()` and friends (cont'd)

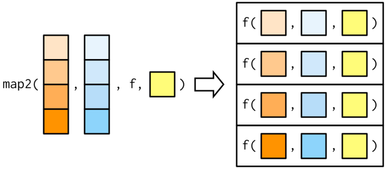
- Both arguments are varied in each call.

```
map2_dbl(xs, ws, weighted.mean)
#> [1]    NA 0.446 0.440 0.417
```

- Additional arguments still go afterwards.

```
map2_dbl(xs, ws, weighted.mean, na.rm = TRUE)
#> [1] 0.326 0.446 0.440 0.417
```



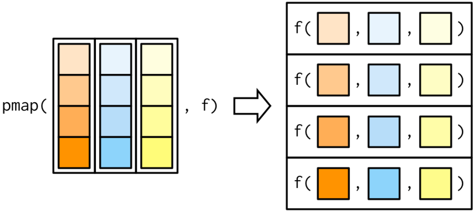


Any number of inputs: `pmap()`

- `map()` and `map2()`, maybe also `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).

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

pmap(params, runif)
#> [[1]]
#> [1] 0.653
#>
#> [[2]]
#> [1] 28.6 96.5
#>
#> [[3]]
#> [1] 106 912 181
```

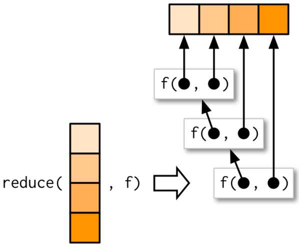


- Note that you can also specify the <type> of the output using `pmap_dbl()`, `pmap_chr()`, `pmap_int()`, and `pmap_lgl()`.

`purrr::reduce()` family

- The next most important family of functionals.
  - Much smaller (two main variants).
  - Powers the map-reduce framework.
- `purrr::reduce()`:
  1. Takes a vector of length `n`.
  2. 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)`.



purrr::reduce() family (cont'd)

- Useful to generalise a function with two arguments to work with any number of inputs.
- **Task**: Find the values that occur in every `element` of a `list`.

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

- Two solutions:

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

```
reduce(value_list, intersect)
#> [1]  1  7 10
```

purrr::accumulate()

```
purrr::accumulate(value_list, intersect)
#> [[1]]
#>  [1]  1  2  2  3  4  4  6  6  7  7  8  8 10 10 10
#>
#> [[2]]
#> [1]  1  2  3  7  8 10
#>
#> [[3]]
#> [1]  1  2  7 10
#>
#> [[4]]
#> [1]  1  7 10

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
```

Predicate functionals

- A **predicate**:
  - Function that returns a single `TRUE` or `FALSE`.
  - E.g., `is.character()`, `is.null()`, or `all()`.
- A **predicate functional** `f(.x, .p)` applies a predicate `.p` to each element of a vector `.x`.
- **Typical predicates** from `purrr` package:
  - `some(.x, .p)`: returns `TRUE` if *any* element matches.
  - `every(.x, .p)`: returns `TRUE` if *all* elements match.
  - `none(.x, .p)`: returns `TRUE` if *no* element matches.
  - `detect(.x, .p)`: returns the *value* of the first match.
  - `detect_index(.x, .p)`: returns the *location* of the first match.
  - `keep(.x, .p)`: *keeps* all matching elements.
  - `discard(.x, .p)`: *drops* all matching elements.

Predicate functionals (cont'd)

```
tbl <- tibble(
  x = 1:3,
  y = c("a", "b", "c")
)
```

```
detect(tbl, is.character)
#> [1] "a" "b" "c"
```

```
keep(tbl, is.character)
#> # A tibble: 3 x 1
#>   y
#>   <chr>
#> 1 a
#> 2 b
#> 3 c
```

```
detect_index(tbl, is.character)
#> [1] 2
```

```
discard(tbl, is.character)
#> # A tibble: 3 x 1
#>       x
#>   <int>
#> 1     1
#> 2     2
#> 3     3
```

Mathematical functionals

Base R provides a useful set:

- `integrate()` finds the area under the curve defined by `f()`
- `uniroot()` finds where `f()` hits zero
- `optimise()` finds the location of the lowest (or highest) value of `f()`

```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
```

```r
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
```

Agenda

Function operators

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

```
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
```

- For {python} users: decorators is just another name!

purrr::safely(): Dealing with failures

- A function modified by `purrr::safely()` always returns a list with two elements:
  1. `result`: the original result.
  2. `error`: 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"
```

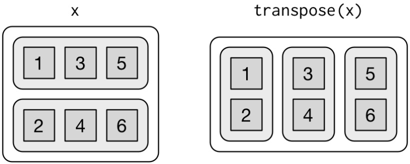
purrr::safely() with purrr::map()

```
values <- list(1, 10, "a")
x <- map(values, safely(log))

str(x)
#> 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"..
```

purrr::list_transpose()

```
x <- purrr::list_transpose(x)
str(x)
#> 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

```
is_ok <- map_lgl(x$error, rlang::is_null)
values[!is_ok]
#> [[1]]
#> [1] "a"
purrr::list_c(x$result[is_ok])
#> [1] 0.0 2.3
```

Two other useful adverbs

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

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

- `purrr::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)
```

memoise::memoise(): Caching computations

- memoise::memoise(): **caches** a function's results.
  - The function remembers previous inputs/returns.
  - Classic CS trade-off of memory versus speed:
  - A memoise'd 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] 4.13
#>    user  system elapsed
#>    0.00    0.00    1.02
system.time(print(slow_fct(1)))
#> [1] 3.65
#>    user  system elapsed
#>    0.01    0.00    1.02
```

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


system.time(print(fast_fct(1)))
#> [1] 6.21
#>    user  system elapsed
#>    0.00    0.00    1.03
system.time(print(fast_fct(1)))
#> [1] 6.21
#>    user  system elapsed
#>    0.02    0.00    0.01
```

memoise::memoise(): Fibonacci series

- Defined recursively:
  - **Initial values**: $f(0) = 0$, $f(1) = 1$,
  - **Definition**: $f(n) = f(n-1) + f(n-2)$.

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

system.time(fib(23))
#>    user  system elapsed
#>    0.01    0.00    0.03
system.time(fib(24))
#>    user  system elapsed
#>    0.03    0.00    0.05
```

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

system.time(fib2(23))
#>    user  system elapsed
#>    0.00    0.00    0.02
system.time(fib2(24))
#>    user  system elapsed
#>       0       0       0
```

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

This slide is Nice! :)

— *Professor Alex*