

# DSFBA: Control Flow and Functions

*Data Science for Business Analytics*

Thibault Vatter

Department of Statistics, Columbia University

05/07/2020

1 Choices

2 Loops

3 Functions

4 Lexical scoping

5 Functional programming

6 Functionals

7 Function operators

- Two primary tools of control flow:
  - ▶ Choices.
  - ▶ Loops.
- Choices:
  - ▶ E.g. `if()`, `ifelse()`, `switch()`.
  - ▶ Allows to run different code depending on the input.
- Loops:
  - ▶ E.g. `for`, `while`, `repeat`.
  - ▶ Allows to repeatedly run code, typically with changing options.
- You might want to have a look at chapter 8 for condition system (messages, warnings, and errors). . .

- The basic idea for if statements:
  - ▶ If condition is TRUE, true\_action is evaluated.
  - ▶ If condition is FALSE, the optional false\_action is evaluated.

```
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 returns a value so that you can assign the results:
  - ▶ Only do that when it fits on one line; otherwise hard to read.

```
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.

```
greet <- function(name, birthday = FALSE) {
  paste0("Hi ", name, if (birthday) " and HAPPY BIRTHDAY")
}

greet("Maria", FALSE)
#> [1] "Hi Maria"

greet("Jaime", TRUE)
#> [1] "Hi Jaime and HAPPY BIRTHDAY"
```

- The condition should evaluate to a single TRUE or FALSE:

```
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
```

- The exception (frequent source of bugs, avoid):
  - ▶ A logical vector of length greater than 1 generates a warning.

```
if (c(TRUE, FALSE)) 1
#> Warning in if (c(TRUE, FALSE)) 1: the condition has length > 1 and
#> only the first element will be used
#> [1] 1
```

# Vectorised if() statements

- if only works with a single TRUE or FALSE.
- What if you have a vector of logical values?
- Answer: ifelse()
  - ▶ Vectorized function with test, yes, and no vectors (recycled).
  - ▶ Missing values propagated into the output.
  - ▶ Advice: use only when yes and no vectors are of the same type.

```
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:

```
dplyr::case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  TRUE ~ as.character(x)
)
#> [1] "1" "2" "3" "4" "fizz" "6" "buzz" "8" "9"
```

- Lets you replace code like:

```
x_option <- function(x) {  
  if (x == "a") {  
    "option 1"  
  } else if (x == "b") {  
    "option 2"  
  } else {  
    stop("Invalid `x` value")  
  }  
}
```

- With:

```
x_option <- function(x) {  
  switch(x,  
    a = "option 1",  
    b = "option 2",  
    stop("Invalid `x` value")  
  )  
}
```



- Last component should always throw an error.
- When multiple inputs share an output:
  - ▶ Use empty right hand sides of =.
  - ▶ Same as C's switch statement.

```
(switch("c", a = 1, b = 2))  
#> NULL
```

```
legs <- function(x) {  
  switch(x,  
    cow = ,  
    horse = ,  
    dog = 4,  
    human = ,  
    chicken = 2,  
    plant = 0,  
    stop("Unknown input")  
  )  
}  
legs("cow")  
#> [1] 4  
legs("dog")  
#> [1] 4
```

- switch() with a numeric x is not recommended.

1 Choices

2 Loops

3 Functions

4 Lexical scoping

5 Functional programming

6 Functionals

7 Function operators

- Two primary tools of control flow:
  - ▶ Choices.
  - ▶ Loops.
- Choices:
  - ▶ E.g. `if()`, `ifelse()`, `switch()`.
  - ▶ Allows to run different code depending on the input.
- Loops:
  - ▶ E.g. `for`, `while`, `repeat`.
  - ▶ Allows to repeatedly run code, typically with changing options.
- You might want to have a look at chapter 8 for condition system (messages, warnings, and errors). . .

- Three main benefits:
  - ▶ Easier to see the intent of your code.
  - ▶ Easier to respond to changes in requirements.
  - ▶ Likely to have fewer bugs.
- Main tools for reducing duplication:
  - ▶ **Functions** identify repeated patterns of code and extract them out into independent pieces.
  - ▶ **Iteration** helps you when you need to do the same thing to multiple inputs.

- Let's compute the median of each column:

```
df <- tibble(a = rnorm(10),  
             b = rnorm(10),  
             c = rnorm(10),  
             d = rnorm(10))  
  
c(median(df$a), median(df$b), median(df$c), median(df$d))  
#> [1] 0.204 -0.355 0.090 -0.258
```

- What's “wrong” here?

- Let's compute the median of each column:

```
df <- tibble(a = rnorm(10),  
             b = rnorm(10),  
             c = rnorm(10),  
             d = rnorm(10))  
  
c(median(df$a), median(df$b), median(df$c), median(df$d))  
#> [1] 0.204 -0.355 0.090 -0.258
```

- What's "wrong" here?

- A better solution:

```
output <- vector("double", ncol(df)) # 1. output  
for (i in seq_along(df)) {           # 2. sequence  
  output[[i]] <- median(df[[i]])      # 3. body  
}  
output  
#> [1] 0.204 -0.355 0.090 -0.258
```

- for loops are used to iterate over items in a vector.

```
for (item in vector) perform_action
```

- For each item in vector, perform\_action is called once; updating the value of item each time.

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

```
#> [1] 1  
#> [1] 2  
#> [1] 3
```

- When iterating over indices, use short names like i, j, or k.
- Important: for assigns the item to the current environment.

```
i <- 100  
for (i in 1:3) {}  
i  
#> [1] 3
```

- Two ways to terminate a for loop early:
  - ▶ next exits the current iteration.
  - ▶ break exits the entire for loop.

```
for (i in 1:10) {  
  if (i < 3)  
    next  
  
  print(i)  
  
  if (i >= 5)  
    break  
}  
#> [1] 3  
#> [1] 4  
#> [1] 5
```



- Common pitfalls to watch out for when using for:
  - ▶ Preallocation.
  - ▶ Iteration over e.g. `1:length(x)`.
- Preallocation:
  - ▶ If you're generating data, preallocate the output.
  - ▶ Otherwise the loop will be very slow.
  - ▶ `vector()` function is helpful.

```
means <- c(1, 50, 20)
out <- vector("list", length(means))
for (i in 1:length(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
```

- Next, beware of iterating over `1:length(x)`, which will fail in unhelpful ways if `x` has length 0.

```
means <- c()
out <- vector("list", length(means))
for (i in 1:length(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
#> Error in rnorm(10, means[[i]]): invalid arguments

# The reason? `:` works with both increasing and decreasing sequences.
1:length(means)
#> [1] 1 0
```

- Use `seq_along(x)` instead.

```
seq_along(means)
#> integer(0)

out <- vector("list", length(means))
for (i in seq_along(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
```

- for loops:
  - ▶ Useful when known in advance the set of values to iterate over.
  - ▶ Otherwise:
    - `while(condition) action:` performs action while condition is TRUE.
    - `repeat(action):` repeats action forever (i.e. until it encounters break).
    - Possible to write any for using while, and any while using repeat, but not the converse.
    - Good practice to use the least-flexible (i.e., simplest) solution to a problem.
- Generally speaking you shouldn't need to use for loops for **data analysis tasks**, we'll see better solutions.

1 Choices

2 Loops

**3 Functions**

4 Lexical scoping

5 Functional programming

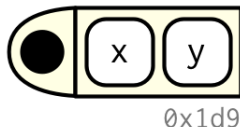
6 Functionals

7 Function operators

- Two **important ideas**:
  - ▶ Functions can be broken down into three components: arguments, body, and environment.
  - ▶ Functions are objects, just as vectors are objects.
- In the following:
  - ▶ **The basics**:
    - How to create functions.
    - The three main components of a function.
    - How can a function exit.
    - Anonymous functions.
    - Lazy evaluation.
    - 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.

- A function has three parts:
  - ▶ The `formals()`, list of arguments controlling how you call the function.
  - ▶ The `body()`, the code inside the function.
  - ▶ The `environment()`, the data structure that determines how the function finds the values associated with the names.

```
f02 <- function(x, y) {  
  # A comment  
  x + y  
}
```



- How those are defined?
  - ▶ Explicitly for the formals and body.
  - ▶ Implicitly for the environment (where the function was defined).

```
formals(f02)
```

```
#> $x
```

```
#>
```

```
#>
```

```
#> $y
```

```
body(f02)
```

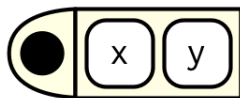
```
#> {
```

```
#>   x + y
```

```
#> }
```

```
environment(f02)
```

```
#> <environment: R_GlobalEnv>
```



0x1d9

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

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

- 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
```



- Most functions exit in one of two ways:
  - ▶ They either return a value, indicating success.
  - ▶ Or they throw an error, indicating failure.
- In the next few slides:
  - ▶ Return values.
    - Implicit versus explicit.
    - Visible versus invisible.
  - ▶ Errors.

- Implicit, where the last evaluated expression is the return value.

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

- Explicit, by calling `return()`.

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

- If a function cannot complete its assigned task, it should throw an error with `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.

- Unlike in many other languages, no special syntax:
  - ▶ Create a function object (with `function`).
  - ▶ Bind it to a name with `<-`.

```
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
```

- In R, function arguments are **lazily evaluated**:
  - ▶ Only evaluated if accessed.
  - ▶ What will this code return?

```
h01 <- function(x) {  
  10  
}  
h01(stop("This is an error!"))
```

- Allows to include expensive computations in function arguments that are only evaluated if needed.

## ... (dot-dot-dot)

- The special argument ... (pronounced dot-dot-dot).
  - ▶ Makes a function take any number of additional arguments.
  - ▶ In other programming languages:
    - This is often called *varargs* (short for variable arguments).
    - A function that uses it is said to be variadic.
- Can pass those additional arguments on 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
```

1 Choices

2 Loops

3 Functions

**4 Lexical scoping**

5 Functional programming

6 Functionals

7 Function operators

- **Scoping:** the act of finding the value associated with a name.
- What does the following code return?

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

g01()
```

- R uses **lexical scoping**<sup>1</sup>:
  - ▶ Looks up the values of names based on how a function is defined, not how it is called.
  - ▶ Follows four primary rules:
    - Name masking
    - Functions versus variables
    - A fresh start
    - Dynamic lookup

---

<sup>1</sup>but possible to override the default rules.



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

- Same applies if a function is defined inside another function:
  - ▶ First, R looks inside the current function.
  - ▶ Then, where that function was defined (and so on, all the way up to the global environment).
  - ▶ Finally, in other loaded packages.
- What does the following code return?

```
x <- 1
g04 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
g04()
```

- Functions are ordinary objects, the same rules apply to them.

```
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.
  - ▶ For function calls, R ignores non-functions when scoping.

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

- But using the same name for different things is best avoided!

- What happens to values between invocations of a function?
- What will happen the first time you run this function?
- What will happen the second time?

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

```
g11()
```

```
g11()
```

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

```
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.

1 Choices

2 Loops

3 Functions

4 Lexical scoping

**5 Functional programming**

6 Functionals

7 Function operators

- To compute the mean of every column:

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

```
output
#> [1]  0.3589 -0.3625  0.0694  0.1539
```

- As a function:

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

```
col_mean(output)
#> [1]  0.3589 -0.3625  0.0694  0.1539
```

# How about other quantities?

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

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

```
col_median(df)  
#> [1] 0.204 -0.355 0.090 -0.258  
col_sd(df)  
#> [1] 1.205 0.679 0.697 0.772
```

- What's "wrong" here?



# A simple “functional”

```
col_summary <- function(df, fun) {  
  out <- vector("double", length(df))  
  for (i in seq_along(df)) {  
    out[i] <- fun(df[[i]])  
  }  
  out  
}  
  
col_summary(df, median)  
#> [1] 0.204 -0.355 0.090 -0.258  
col_summary(df, mean)  
#> [1] 0.3589 -0.3625 0.0694 0.1539
```

## ■ 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.

- 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:
  - ▶ 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.

- 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.

```
f1 <- function(x) x
l1 <- list(
  mean,
  sd,
  median
)
y <- rnorm(1e1)
sapply(l1, function(f) f(y))
#> [1] -0.1050  0.7437 -0.0493
```

- 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*.

- 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	Function
Vector	Regular function	Function factory
Function	Functional	Function operator

1 Choices

2 Loops

3 Functions

4 Lexical scoping

5 Functional programming

**6 Functionals**

7 Function operators



*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()`.

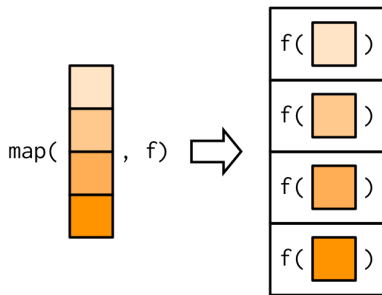
```
randomise <- function(f) f(runif(1e3))  
randomise(mean)  
#> [1] 0.497  
randomise(mean)  
#> [1] 0.487  
randomise(sum)  
#> [1] 497
```

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

```
library(purrr)
```

- 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()`.

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



- 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.

- `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"
```

- `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.

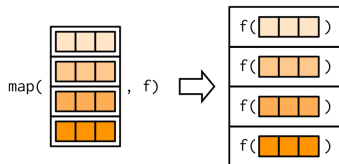
```
map_lgl(mtcars, is.double)  
#>   mpg   cyl  disp    hp  drat    wt   qsec    vs    am gear carb  
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

- 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.



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

```
map_dbl(1:2, function(x) c(x, x))  
#> Error: 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)`.

- 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] 0.22 0.089  
#> $ : num [1:2] 0.131 0.879  
#> $ : num [1:2] 0.652 0.0281
```

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

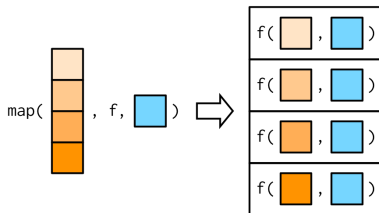


- To pass along additional arguments, use an anonymous function.

```
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] 0.131 0.131 0.131 0.131
```

```
map_dbl(x, ~ plus(.x, runif(1)))  
#> [1] 0.158 0.473 0.751 0.840
```

## ■ 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	<code>map()</code>	<code>map_lgl()</code> , ...	<code>modify()</code>	<code>walk()</code>
Two arguments	<code>map2()</code>	<code>map2_lgl()</code> , ...	<code>modify2()</code>	<code>walk2()</code>
One argument + index	<code>imap()</code>	<code>imap_lgl()</code> , ...	<code>imodify()</code>	<code>iwalk()</code>
N arguments	<code>pmap()</code>	<code>pmap_lgl()</code> , ...	—	<code>pwalk()</code>

# Two inputs: map2() and friends

- How do we find the vector of weighted means?

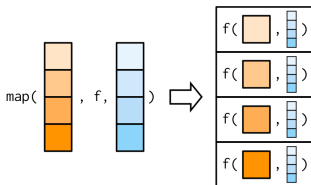
```
xs <- map(1:8, ~ runif(10))  
xs[[1]][[1]] <- NA  
ws <- map(1:8, ~ rpois(10, 5) + 1)
```

- Use map\_dbl() to compute the unweighted means.

```
map_dbl(xs, mean)  
#> [1] NA 0.428 0.524 0.568 0.583 0.571 0.530 0.465
```

- 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 length
```

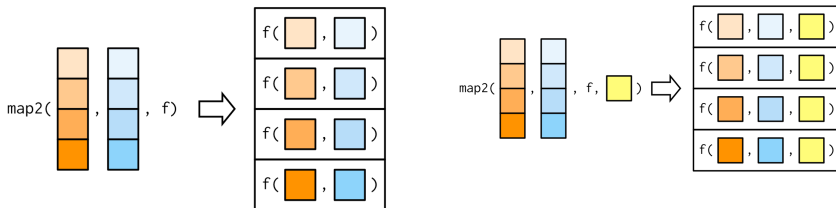


- Both arguments are varied in each call.

```
map2_dbl(xs, ws, weighted.mean)
#> [1] NA 0.433 0.515 0.565 0.571 0.668 0.539 0.433
```

- Additional arguments still go afterwards.

```
map2_dbl(xs, ws, weighted.mean, na.rm = TRUE)
#> [1] 0.378 0.433 0.515 0.565 0.571 0.668 0.539 0.433
```

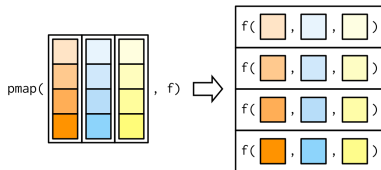


# Any number of inputs: pmap()

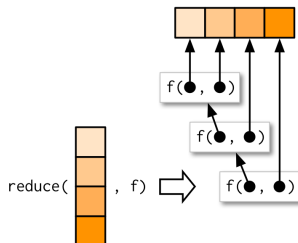
- `map()` and `map2()`... `map3()`, `map4()`, `map5()`?
- Instead, there is `pmap()`:
  - ▶ Supply it a single list, which contains any number of arguments.
  - ▶ In most cases, a list of equal-length vectors (e.g., a data frame).

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

```
pmap(params, runif)  
#> [[1]]  
#> [1] 0.0704  
#>  
#> [[2]]  
#> [1] 73.5 16.7  
#>  
#> [[3]]  
#> [1] 393 423 449
```



- 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)`.



- 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 = TRUE))
str(l)
#> List of 4
#> $ : int [1:15] 3 7 10 2 7 3 4 5 8 2 ...
#> $ : int [1:15] 6 8 3 4 3 10 9 8 1 2 ...
#> $ : int [1:15] 6 8 8 3 3 2 9 2 10 9 ...
#> $ : int [1:15] 6 10 1 8 6 10 6 8 3 2 ...
```

- Two solutions:

```
out <- l[[1]]
out <- intersect(out, l[[2]])
out <- intersect(out, l[[3]])
out <- intersect(out, l[[4]])
out
#> [1] 3 10 2 8 6
```

```
reduce(l, intersect)
#> [1] 3 10 2 8 6
```

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

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
```



## ■ A predicate:

- ▶ Function that returns a single TRUE or FALSE.
- ▶ E.g., `is.character()`, `is.null()`, or `all()`.
- ▶ **Matches** a vector if it returns TRUE.

## ■ A predicate functional:

- ▶ `f(x, p)` applies a predicate `p` to each element of a vector `x`.
- ▶ `some()/every()`: TRUE if *any/all* element matches.
- ▶ `detect()/detect_index()`: *value/location* of the first match.
- ▶ `keep()/discard()`: *keeps/drops* all matching elements.

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

```
detect(df, is.character)
```

```
#> [1] "a" "b" "c"
```

```
keep(df, is.character)
```

```
#> # A tibble: 3 x 1
```

```
#>   y
```

```
#>   <chr>
```

```
#> 1 a
```

```
#> 2 b
```

```
#> 3 c
```

```
detect_index(df, is.character)
```

```
#> [1] 2
```

```
discard(df, is.character)
```

```
#> # A tibble: 3 x 1
```

```
#>   x
```

```
#>   <int>
```

```
#> 1     1
```

```
#> 2     2
```

```
#> 3     3
```

```
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
```

1 Choices

2 Loops

3 Functions

4 Lexical scoping

5 Functional programming

6 Functionals

**7 Function operators**

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

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

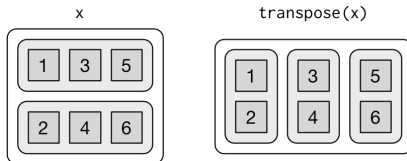
```
safe_log <- safely(log)
str(safe_log(10))
#> List of 2
#> $ result: num 2.3
#> $ error : NULL
str(safe_log("a"))
#> List of 2
#> $ result: NULL
#> $ error :List of 2
#> ..$ message: chr "non-numeric argument to mathematical function"
#> ..$ call : language .Primitive("log")(x, base)
#> ..- attr(*,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

```
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"..
```



```
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
```



- `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)
```

- 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.

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

```
system.time(print(slow_fct(1)))  
#> [1] 9.57  
#>   user system elapsed  
#> 0.001 0.000 1.002  
system.time(print(slow_fct(1)))  
#> [1] 2.39  
#>   user system elapsed  
#> 0.006 0.000 1.007
```

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

```
system.time(print(fast_fct(1)))  
#> [1] 4.07  
#>   user system elapsed  
#> 0.001 0.000 1.002  
system.time(print(fast_fct(1)))  
#> [1] 4.07  
#>   user system elapsed  
#> 0.028 0.000 0.027
```

- Defined recursively:

- ▶  $f(0) = 0, f(1) = 1,$
- ▶ And then  $f(n) = f(n-1) + f(n-2).$

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

```
system.time(fib(23))  
#>    user system elapsed  
#> 0.042 0.000 0.042  
system.time(fib(24))  
#>    user system elapsed  
#> 0.068 0.000 0.068
```

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

```
system.time(fib2(23))  
#>    user system elapsed  
#> 0.037 0.000 0.037  
system.time(fib2(24))  
#>    user system elapsed  
#> 0.002 0.000 0.001
```

- An example of **dynamic programming**:

- ▶ Complex problem broken down into overlapping subproblems.
- ▶ Remembering the results of a subproblem considerably improves performance.