MONASH University

MONASH BUSINESS SCHOOL

ETC4500/ETC5450 Advanced R programming

Week 5: Reproducible environments and functional programming

`arp.numbat.space`

Outline

1 Assignments
2 Reproducible environments
3 Programming paradigms
4 Functional programming
5 Functional problem solving

Outline

1 Assignments
2 Reproducible environments
3 Programming paradigms
4 Functional programming
5 Functional problem solving

Assignment 1

- Keep working on your package!
- Final version due on 31 May 2024

Assignment 2

- About debugging and profiling
- Available on GitHub Classroom today!
- Due 19 April 2024

Outline

1 Assignments

2 Reproducible environments

3 Programming paradigms

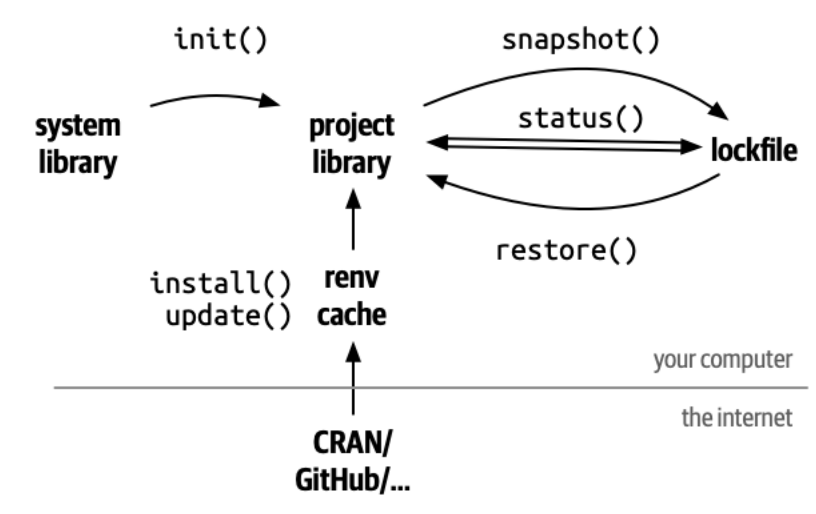
4 Functional programming

5 Functional problem solving

Reproducible environments

- To ensure that your code runs the same way on different machines and at different times, you need the computing environment to be the same.
  1. Operating system
  2. System components
  3. R version
  4. R packages
- Solutions for 1–4: Docker, Singularity, `containerit`, `rang`
- Solutions for 4: `packrat`, `checkpoint`, `renv`

renv package



renv package

- `renv::init()` : initialize a new project with a new environment. Adds:
  - `renv/library` contains all packages used in project
  - `renv.lock` contains metadata about packages used in project
  - `.Rprofile` run every time R starts.
- `renv::snapshot()` : save the state of the project to `renv.lock`.
- `renv::restore()` : restore the project to the state saved in `renv.lock`.

renv package

- renv uses a package cache so you are not repeatedly installing the same packages in multiple projects.
- `renv::install()` can install from CRAN, Bioconductor, GitHub, Gitlab, Bitbucket, etc.
- `renv::update()` gets latest versions of all dependencies from wherever they were installed from.
- Only R packages are supported, not system dependencies, and not R itself.
- renv is not a replacement for Docker or Singularity.
- `renv::deactivate(clean = TRUE)` will remove the renv environment.

Exercises

5. Add renv to your Assignment 1 project. Make sure the packages included are the latest CRAN versions of all packages.

Outline

1. Assignments
2. Reproducible environments
3. Programming paradigms
4. Functional programming
5. Functional problem solving

Programming paradigms

R code is typically structured using these paradigms:

- Functional programming
- Object-oriented programming
- Literate programming
- Reactive programming

Often several paradigms used together to solve a problem.

Programming paradigms

Functional programming (W5; today!)

- Functions are created and used like any other object.
- Output should only depend on the function's inputs.

Programming paradigms

Functional programming (W5; today!)

- Functions are created and used like any other object.
- Output should only depend on the function's inputs.

Object-oriented programming (W6)

- Functions are associated with object types.
- Methods of the same 'function' produce object-specific output.

Programming paradigms

Literate programming (W7)

- Natural language is interspersed with code.
- Aimed at prioritising documentation/comments.
- Now used to create reproducible reports/documents.

Programming paradigms

Literate programming (W7)

- Natural language is interspersed with code.
- Aimed at prioritising documentation/comments.
- Now used to create reproducible reports/documents.

Reactive programming (W7)

- Objects are expressed using code based on inputs.
- When inputs change, the object's value updates.

Outline

1 Assignments

2 Reproducible environments

3 Programming paradigms

4 Functional programming

5 Functional problem solving

Functional programming

R is commonly considered a ‘functional’ programming language - and so far we have used functional programming.

```
square <- function(x) {
  return(x^2)
}
square(8)
```

```
[1] 64
```

The `square` function is an object like any other in R.

Functions are objects

R functions can be printed,

```r
print(square)
```

```
function(x) {
  return(x^2)
}
```

Functions are objects

R functions can be printed,

```r
print(square)
```

```
function(x) {
  return(x^2)
}
```

inspected,

```r
formals(square)
```

```
$x
```

Functions are objects

put in a list,

```r
my_functions <- list(square, sum, min, max)
my_functions
```

```
[[1]]
function(x) {
  return(x^2)
}

[[2]]
function (..., na.rm = FALSE)  .Primitive("sum")

[[3]]
function (..., na.rm = FALSE)  .Primitive("min")

[[4]]
function (..., na.rm = FALSE)  .Primitive("max")
```

Functions are objects

used within lists,

```
my_functions[[1]](8)
```

```
[1] 64
```

Functions are objects

used within lists,

```
my_functions[[1]](8)
```

```
[1] 64
```

but they can’t be subsetted!

```
square$x
```

```
Error in square$x: object of type 'closure' is not subsettable
```

Handling input types

Functional programming handles different input types using control flow. The same code is ran regardless of object type.

```
square <- function(x) {
  if(!is.numeric(x)) {
    stop("`x` needs to be numeric")
  }
  return(x^2)
}
```

Handling input types

Functional programming handles different input types using control flow. The same code is ran regardless of object type.

```
square <- function(x) {
  if(!is.numeric(x)) {
    stop("`x` needs to be numeric")
  }
  return(x^2)
}
```

Next class…

We will see object-oriented programming, which handles different input types using different functions (methods)!

What are functions?

A function is comprised of three components:

- The arguments/inputs (`formals()`)
- The body/code (`body()`)
- The environment (`environment()`)

What are functions?

A function is comprised of three components:

- The arguments/inputs (`formals()`)
- The body/code (`body()`)
- The environment (`environment()`)

Your turn!

Use these functions to take a closer look at `square()`.
Try modifying the function's formals/body/env with `<-`.

Functional programming

Since functions are like any other object, they can also be:

- **inputs** to functions

> Extensible design with function inputs
>
> Using function inputs can improve your package's design! Rather than limiting users to a few specific methods, allow them to use and write any method with functions.

Function arguments

Consider a function which calculates accuracy measures:

```
accuracy <- function(e, measure, ...) {
  if (measure == "mae") {
    mean(abs(e), ...)
  } else if (measure == "rmse") {
    sqrt(mean(e^2, ...))
  } else {
    stop("Unknown accuracy measure")
  }
}
```

Improving the design

This function is limited to only computing MAE and RMSE.

Function arguments

Using function operators allows any measure to be used.

```
MAE <- function(e, ...) mean(abs(e), ...)
RMSE <- function(e, ...) sqrt(mean(e^2, ...))
accuracy <- function(e, measure, ...) {
  ???
}
accuracy(rnorm(100), measure = RMSE)
```

Your turn!

Complete the accuracy function to calculate accuracy statistics based on the function passed in to `measure`.

Functional programming

Since functions are like any other object, they can also be:

- **inputs** to functions
- **outputs** of functions

Functions making functions?

These functions are known as *function factories*.
Where have you seen a function that creates a function?

Function factories

Let's generalise `square()` to raise numbers to any power.

```r
power <- function(x, exp) {
  x^exp
}
power(8, exp = 2)
```

```
[1] 64
```

```r
power(8, exp = 3)
```

```
[1] 512
```

> **Starting a factory**
>
> What if the function returned a function instead?

Function factories

```
power_factory <- function(exp) {
  # R is lazy and won't look at exp unless we ask it to
  force(exp)
  # Return a function, which finds exp from this environment
  function(x) {
    x^exp
  }
}
square <- power_factory(exp = 2)
square(8)
```

```
[1] 64
```

Function factories

```r
power_factory <- function(exp) {
  # R is lazy and won't look at exp unless we ask it to
  force(exp)
  # Return a function, which finds exp from this environment
  function(x) {
    x^exp
  }
}
square <- power_factory(exp = 2)
square(8)
```

```
[1] 64
```

```r
cube <- power_factory(exp = 3)
cube(8)
```

```
[1] 512
```

Function factories

Consider this function to calculate plot breakpoints of vectors.

```
breakpoints <- function(x, n.breaks) {
  seq(min(x), max(x), length.out = n.breaks)
}
```

Your turn!

Convert this function into a function factory.

Is it better to create functions via `x` or `n.breaks`?

Outline

1. Assignments
2. Reproducible environments
3. Programming paradigms
4. Functional programming
5. Functional problem solving

Split, apply, combine

Many problems can be simplified/solved using this process:

- split (break the problem into smaller parts)
- apply (solve the smaller problems)
- combine (join solved parts to solve original problem)

Split, apply, combine

Many problems can be simplified/solved using this process:

- split (break the problem into smaller parts)
- apply (solve the smaller problems)
- combine (join solved parts to solve original problem)

This technique applies to both

- writing functions (rewriting a function into sub-functions)
- working with data (same function across groups or files)

data |> group_by() |> summarise()

An example of split-apply-combine being used to work with data is when `group_by()` and `summarise()` are used together.

data |> group_by() |> summarise()

An example of split-apply-combine being used to work with data is when `group_by()` and `summarise()` are used together.

- split: `group_by()` splits up the data into groups
- apply: your `summarise()` code calculates a single value
- combine: `summarise()` combines the results into a vector

data |> group_by() |> summarise()

An example of split-apply-combine being used to work with data is when `group_by()` and `summarise()` are used together.

- split: `group_by()` splits up the data into groups
- apply: your `summarise()` code calculates a single value
- combine: `summarise()` combines the results into a vector

```
library(dplyr)
mtcars |>
  group_by(cyl) |>
  summarise(mean(mpg))
```

```
# A tibble: 3 x 2
    cyl `mean(mpg)`
  <dbl>       <dbl>
1     4        26.7
2     6        19.7
3     8        15.1
```

Split-apply-combine for vectors and lists

The same idea can be used for calculations on vectors.

Split-apply-combine for vectors and lists

The same idea can be used for calculations on vectors.

There are two main implementations we consider:

- base R: The `*apply()` functions
- purrr: The `map*()` functions

Split-apply-combine for vectors and lists

The same idea can be used for calculations on vectors.

There are two main implementations we consider:

- base R: The `*apply()` functions
- purrr: The `map*()` functions

We will use purrr and but I'll also share the base R equivalent.

for or map?

Let's `square()` a vector of numbers with a for loop.

```
x <- c(1, 3, 8)
x2 <- numeric(length(x))
for (i in seq_along(x)) {
  x2[i] <- square(x[i])
}
x2
```

```
[1]  1  9 64
```

for or map?

Let's `square()` a vector of numbers with a for loop.

```
x <- c(1, 3, 8)
x2 <- numeric(length(x))
for (i in seq_along(x)) {
  x2[i] <- square(x[i])
}
x2
```

```
[1]  1  9 64
```

Vectorisation?

Of course `square()` is vectorised, so we should use `square(x)`. Other functions like `lm()` or `read.csv()` are not!

for or map?

Instead using `map()` we get…

```
library(purrr)
x <- c(1, 3, 8)
map(x, square) # lapply(x, square)
```

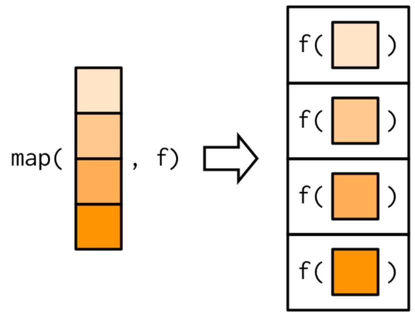
```
[[1]]
[1] 1

[[2]]
[1] 9

[[3]]
[1] 64
```

Mapping vectors

The same result, but it has been combined differently!



Mapping vectors

To combine the results into a vector rather than a list, we instead use `map_vec()` to combine results into a vector.

```
library(purrr)
x <- c(1, 3, 8)
map_vec(x, square) # vapply(x, square, numeric(1L))
```

```
[1]  1  9 64
```

for or map

Advantages of map

- Less coding (less bugs!)
- Easier to read and understand.

for or map

Advantages of map

- Less coding (less bugs!)
- Easier to read and understand.

Disadvantages of map

- Less control over loop
- Cannot solve sequential problems

Functional mapping

Recall `group_by()` and `summarise()` from dplyr:

```
mtcars |>
  group_by(cyl) |>
  summarise(mean(mpg))
```

🔥 Your turn!

Use `split()` and `map_vec()` to achieve a similar result.

Hint: `split(mtcars$mpg, mtcars$cyl)` *creates a list that splits* `mtcars$mpg` *by each value of* `mtcars$cyl`*.*

Anonymous mapper functions

Suppose we want to separately model `mpg` for each `cyl`.

```
lm(mpg ~ disp + hp + drat + wt, mtcars[mtcars$cyl == 4,])
lm(mpg ~ disp + hp + drat + wt, mtcars[mtcars$cyl == 6,])
lm(mpg ~ disp + hp + drat + wt, mtcars[mtcars$cyl == 8,])
```

Anonymous mapper functions

We can split the data by `cyl` with `split()`,

```
mtcars_cyl <- split(mtcars, mtcars$cyl)
```

but `map(mtcars_cyl, lm, mpg ~ disp + hp + drat + wt)` won't work - why?

Anonymous mapper functions

We can split the data by `cyl` with `split()`,

```
mtcars_cyl <- split(mtcars, mtcars$cyl)
```

but `map(mtcars_cyl, lm, mpg ~ disp + hp + drat + wt)` won't work - why?

> **! Difficult to map**
>
> Using `map(mtcars_cyl, lm)` will apply `lm(mtcars_cyl[i])`. The mapped vector is always used as the first argument!

Anonymous mapper functions

We can write our own functions!

```r
mtcars_lm <- function(.) lm(mpg ~ disp + hp + drat + wt, data = .)
map(mtcars_cyl, mtcars_lm)
```

```
$`4`

Call:
lm(formula = mpg ~ disp + hp + drat + wt, data = .)

Coefficients:
(Intercept)         disp           hp         drat           wt
    52.5195      -0.0629      -0.0760      -1.4422      -3.1001


$`6`

Call:
```

Anonymous mapper functions

Or use `~ body` to create anonymous functions.

```
# lapply(mtcars_cyl, \(.) lm(mpg ~ disp + hp + drat + wt, data = .))
map(mtcars_cyl, ~ lm(mpg ~ disp + hp + drat + wt, data = .))
```

```
$`4`

Call:
lm(formula = mpg ~ disp + hp + drat + wt, data = .)

Coefficients:
(Intercept)         disp           hp         drat           wt
    52.5195      -0.0629      -0.0760      -1.4422      -3.1001


$`6`

Call:
```

Mapping mapping mapping

How would you then get the coefficients from all 3 models?

```r
# mtcars_cyl |> lapply(\(.) lm(mpg ~ disp + hp + drat + wt, data = .))
mtcars_cyl |>
  map(~ lm(mpg ~ disp + hp + drat + wt, data = .))
```

Mapping mapping mapping

How would you then get the coefficients from all 3 models?

```r
# mtcars_cyl |> lapply(\(.) lm(mpg ~ disp + hp + drat + wt, data = .))
mtcars_cyl |>
  map(~ lm(mpg ~ disp + hp + drat + wt, data = .))
```

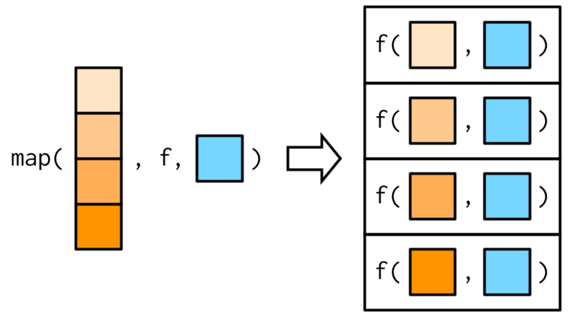
Solution

```r
# lapply(mtcars_cyl, \(.) lm(mpg ~ disp + hp + drat + wt, data = .))
mtcars_cyl |>
  map(~ lm(mpg ~ disp + hp + drat + wt, data = .)) |>
  map(coef)
```

```
$`4`
(Intercept)        disp          hp        drat          wt
    52.5195     -0.0629     -0.0760     -1.4422     -3.1001
```

Mapping arguments

Any arguments after your function are passed to all functions.



Mapping arguments

This works by passing through ... to the function.

```r
x <- list(1:5, c(1:10, NA))
map_dbl(x, ~ mean(.x, na.rm = TRUE))
```

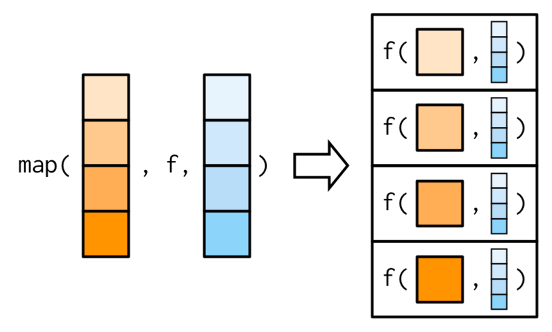
```
[1] 3.0 5.5
```

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

```
[1] 3.0 5.5
```

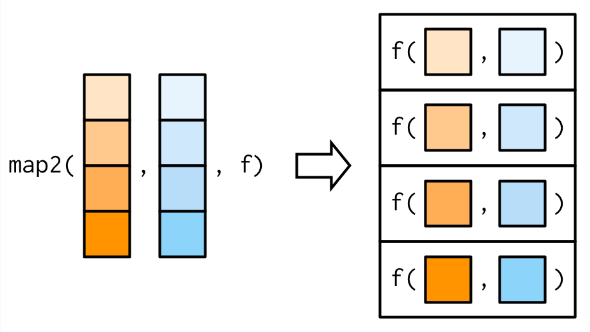
Mapping arguments

These additional arguments are not decomposed / mapped.



Mapping multiple arguments

It is often useful to map multiple arguments.



Mapping multiple arguments

```r
xs <- map(1:8, ~ ifelse(runif(10) > 0.8, NA, runif(10)))
map_vec(xs, mean, na.rm = TRUE)
```

```
[1] 0.516 0.458 0.478 0.555 0.592 0.422 0.396 0.517
```

Mapping multiple arguments

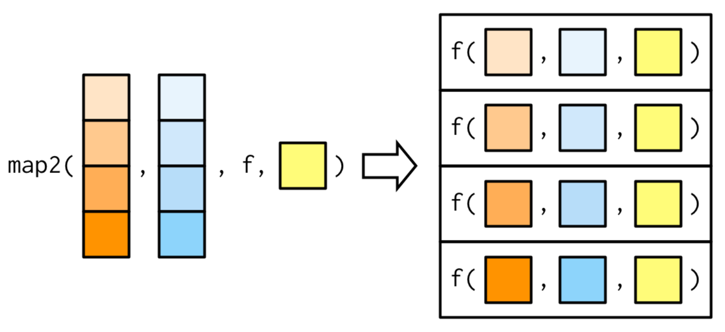
```r
xs <- map(1:8, ~ ifelse(runif(10) > 0.8, NA, runif(10)))
map_vec(xs, mean, na.rm = TRUE)
```

```
[1] 0.516 0.458 0.478 0.555 0.592 0.422 0.396 0.517
```

```r
ws <- map(1:8, ~ rpois(10, 5) + 1)
map2_vec(xs, ws, weighted.mean, na.rm = TRUE)
```

```
[1] 0.546 0.457 0.455 0.541 0.621 0.391 0.338 0.559
```

Mapping multiple arguments



Mapping many arguments

It is also possible to map any number of inputs with `pmap`.

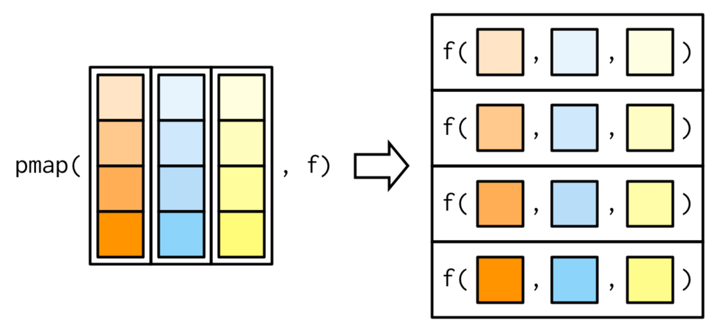
```
n <- 1:3
min <- c(0, 10, 100)
max <- c(1, 100, 1000)
pmap(list(n, min, max), runif) # .mapply(runif, list(n, min, max), list())
```

```
[[1]]
[1] 0.0654

[[2]]
[1] 27.4 64.3

[[3]]
[1] 167 995 733
```

Mapping many arguments



Parallel mapping

Split-apply-combine problems are *embarrassingly parallel*.

Parallel mapping

Split-apply-combine problems are *embarrassingly parallel*.

The furrr package (future + purrr) makes it easy to use `map()` in parallel, providing `future_map()` variants.

```
library(furrr)
plan(multisession, workers = 4)
future_map_dbl(xs, mean, na.rm = TRUE)
```

```
[1] 0.516 0.458 0.478 0.555 0.592 0.422 0.396 0.517
```

```
future_map2_dbl(xs, ws, weighted.mean, na.rm = TRUE)
```

```
[1] 0.546 0.457 0.455 0.541 0.621 0.391 0.338 0.559
```

Reduce vectors to single values

Sometimes you want to collapse a vector, reducing it to a single value. `reduce()` always returns a vector of length 1.

```
x <- sample(1:100, 10)
x
```

```
 [1] 18 49 60 75 97 30 78 34 65 70
```

```
sum(x)
```

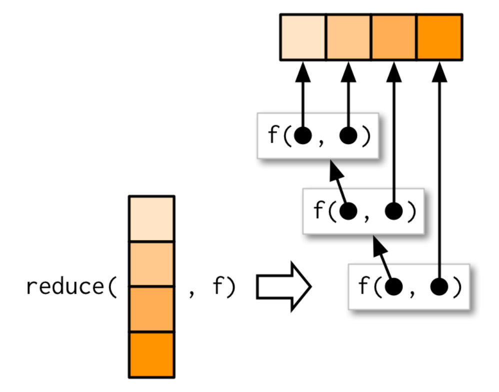
```
[1] 576
```

```
# Alternative to sum()
reduce(x, `+`) # Reduce(`+`, x)
```

```
[1] 576
```

Reduce vectors to single values

The result from the function is re-used as the first argument.



Reduce vectors to single values

Your turn!

We're studying the letters in 3 bowls of alphabet soup.

Reduce vectors to single values

Your turn!

We're studying the letters in 3 bowls of alphabet soup.

Use `reduce()` to find the letters were in all bowls of soup!

Are all letters found in the soups?

```
alphabet_soup <- map(c(10,24,13), sample, x=letters, replace=TRUE)
alphabet_soup
```

```
[[1]]
 [1] "v" "a" "w" "w" "n" "d" "u" "c" "s" "o"

[[2]]
 [1] "o" "i" "m" "e" "i" "a" "p" "u" "f" "o" "e" "w" "t" "z" "n" "o" "p"
[18] "j" "v" "r" "e" "z" "r" "j"
```

Functional adverbs

purrr also offers many *adverbs*, which modify a function.

Capturing conditions

- `possibly(.f, otherwise)`: If the function errors, it will return `otherwise` instead.
- `safely(.f)`: The function now returns a list with 'result' and 'error', preventing errors.
- `quietly(.f)`: Any conditions (messages, warnings, printed output) are now captured into a list.

Functional adverbs

purrr also offers many *adverbs*, which modify a function.

Changing results

- `negate(.f)` will return `!result`.

Chaining functions

- `compose(...)` will chain functions together like a chain of piped functions.

Functional adverbs

purrr also offers many *adverbs*, which modify a function.

Functions modifying functions?

These functions are all *function factories*!
More specifically they are known as *function operators* since both the input and output is a function.
`memoise::memoise()` is also a *function operator*.