Iteration and Functionals with Base R and purrr

Outline

Importance of Iteration in Data Science

Basic R loops/control structures (for, if else, while, etc.)

Base R functionals

purrr

- map() and friends!
- Others that are useful!

Iteration for Data Science

In the modeling part of this course sequence (really for every part), it will be important to create concise and readable code!

Part of that is not repeating your code to do a similar task (DRY principle)

Iteration - tool for reducing duplication when you need to do the same thing to multiple inputs

Imperative Programming (loops)

Functional Programming (functionals)

Functional programming is preferred over imperative programming in R because it is a functional programming language! (This may be really weird for some of you!)

Basic Looping/Control

Task: I want to square every element of a numeric vector!

```
vec <- c(1,2,3,4)
```

Because R is so awesome: `vec^2` `[1] 1 4 9 16`

What if it wasn't so awesome? for loop!

```
for (variable in vector) {

}
```

```
square_vec <- vector(mode = "numeric",
                     length = length(vec))

for (i in seq_along(vec)) {
    square_vec[i] <- vec[i]^2
}
```

Basic Looping/Control

Other basic control flows:

What if you don’t know how many iterations you need?

```
while (condition) {

}
```

Not iteration-based but what if the direction of your code depends on a condition?

```
if (condition) {

} else {

}
```

Functionals

In R, for loops have a really bad rap!

A common use of functionals is as an alternative to for loops. For loops have a bad rap in R because many people believe they are slow[37], but the real downside of for loops is that they're very flexible: a loop conveys that you're iterating, but not what should be done with the results. Just as it's better to use `while` than `repeat`, and it's better to use `for` than `while` (Section 5.3.2), it's better to use a functional than `for`. Each functional is tailored for a specific task, so when you recognise the functional you immediately know why it's being used.

Functional - In general, a functional is a function that takes in a function and outputs a vector

Base R functionals: apply(), lapply(), sapply(), vapply(), integrate(), optim(), etc.

Advanced R, Hadley Wickham

Functionals

`apply()`: Apply a function over matrix or array margins

```
mat
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

```
apply(mat, 1, sum)
```

```
[1]  6 15 24
```

```
apply(mat, 2, sum)
```

```
[1] 12 15 18
```

`lapply()`: "l-apply" (list apply) list in and list out

```
lapply(vec, function(x) x^2)
```

anonymous function

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

[[2]]
[1] 4

[[3]]
[1] 9

[[4]]
[1] 16
```

Functionals

`sapply()`: "s-apply" list in, simplifies output (guesses output type)

```
sapply(vec, function(x) x^2)
```

```
[1]  1  4  9 16
```

Another example:

```
df <- tibble(
    a = rnorm(10),
    b = rnorm(10),
    c = rnorm(10),
    d = rnorm(10)
)
```

```
sapply(df, mean)
```

```
          a           b           c           d
-0.1198698 -0.0195202 -0.5088570  0.1653908
```

purrr

purrr is the package in the tidyverse that provides functional programming tools for doing data analysis!

purrr functionals generally do the same thing as the apply family of functions, but provide more consistency!

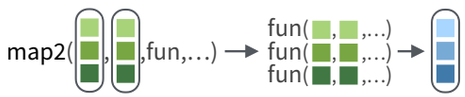
They also provide functionals that don't exist in base R (outside of what we'll see)!

Apply functions with purrr : : **CHEAT SHEET**

Apply Functions

Map functions apply a function iteratively to each element of a list or vector.

map(.x, .f, …) Apply a function to each element of a list or vector. *map(x, is.logical)*



map2(.x, ,y, .f, …) Apply a function to pairs of elements from two lists, vectors. *map2(x, y, sum)*

pmap(.l, .f, …) Apply a function to groups of elements from list of lists, vectors. *pmap(list(x, y, z), sum, na.rm = TRUE)*

invoke_map(.f, .x = list(NULL), …, .env=NULL) Run each function in a list. Also **invoke**. *l <- list(var, sd); invoke_map(l, x = 1:9)*

lmap(.x, .f, ...) Apply function to each list-element of a list or vector.
imap(.x, .f, ...) Apply .f to each element of a list or vector and its index.

OUTPUT

map(), map2(), pmap(), imap and **invoke_map** each return a list. Use a suffixed version to return the results as a specific type of flat vector, e.g. **map2_chr**, **pmap_lgl**, etc.

Use **walk**, **walk2**, and **pwalk** to trigger side effects. Each return its input invisibly.

function	returns
map	list
map_chr	character vector
map_dbl	double (numeric) vector
map_dfc	data frame (column bind)
map_dfr	data frame (row bind)
map_int	integer vector
map_lgl	logical vector
walk	triggers side effects, returns the input invisibly

SHORTCUTS - within a purrr function:

"name" becomes **function(x) x[["name"]]**, e.g. *map(l, "a")* extracts *a* from each element of *l*

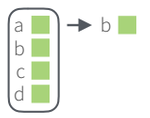
~ .x becomes **function(x) x**, e.g. *map(l, ~ 2 +.x)* becomes *map(l, function(x) 2 + x)*

~ .x .y becomes **function(.x, .y) .x .y**, e.g. *map2(l, p, ~ .x +.y)* becomes *map2(l, p, function(l, p) l + p)*

~ ..1 ..2 etc becomes **function(..1, ..2,** etc**) ..1 ..2** etc, e.g. *pmap(list(a, b, c), ~ ..3 + ..1 - ..2)* becomes *pmap(list(a, b, c), function(a, b, c) c + a - b)*

Work with Lists

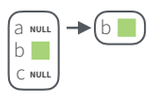
FILTER LISTS



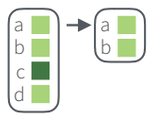
pluck(.x, ..., .default=NULL) Select an element by name or index, *pluck(x,"b")* ,or its attribute with **attr_getter**. *pluck(x,"b",attr_getter("n"))*

keep(.x, .p, …) Select elements that pass a logical test. *keep(x, is.na)*

discard(.x, .p, …) Select elements that do not pass a logical test. *discard(x, is.na)*

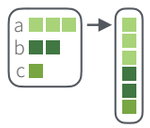


compact(.x, .p = identity) Drop empty elements. *compact(x)*

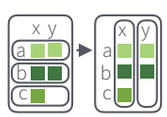


head_while(.x, .p, …) Return head elements until one does not pass. Also **tail_while**. *head_while(x, is.character)*

RESHAPE LISTS



flatten(.x) Remove a level of indexes from a list. Also **flatten_chr**, **flatten_dbl**, **flatten_dfc**, **flatten_dfr**, **flatten_int**, **flatten_lgl**. *flatten(x)*



transpose(.l, .names = NULL) Transposes the index order in a multi-level list. *transpose(x)*

SUMMARISE LISTS

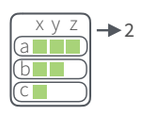
every(.x, .p, …) Do all elements pass a test? *every(x, is.character)*

some(.x, .p, …) Do some elements pass a test? *some(x, is.character)*

has_element(.x, .y) Does a list contain an element? *has_element(x, "foo")*

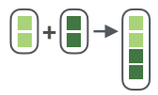
detect(.x, .f, ..., .right=FALSE, .p) Find first element to pass. *detect(x, is.character)*

detect_index(.x, .f, ..., .right = FALSE, .p) Find index of first element to pass. *detect_index(x, is.character)*

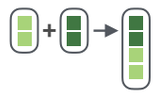


vec_depth(x) Return depth (number of levels of indexes). *vec_depth(x)*

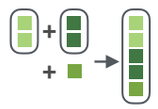
JOIN (TO) LISTS



append(x, values, after = length(x)) Add to end of list. *append(x, list(d = 1))*

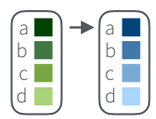


prepend(x, values, before = 1) Add to start of list. *prepend(x, list(d = 1))*

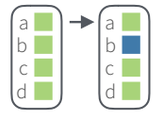


splice(…) Combine objects into a list, storing S3 objects as sub-lists. *splice(x, y, "foo")*

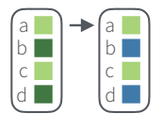
TRANSFORM LISTS



modify(.x, .f, ...) Apply function to each element. Also **map**, **map_chr**, **map_dbl**, **map_dfc**, **map_dfr**, **map_int**, **map_lgl**. *modify(x, ~.+ 2)*



modify_at(.x, .at, .f, ...) Apply function to elements by name or index. Also **map_at**. *modify_at(x, "b", ~.+ 2)*

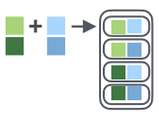


modify_if(.x, .p, .f, ...) Apply function to elements that pass a test. Also **map_if**. *modify_if(x, is.numeric,~.+2)*

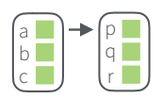
modify_depth(.x,.depth,.f,...) Apply function to each element at a given level of a list. *modify_depth(x, 1, ~.+ 2)*

WORK WITH LISTS

array_tree(array, margin = NULL) Turn array into list. Also **array_branch**. *array_tree(x, margin = 3)*

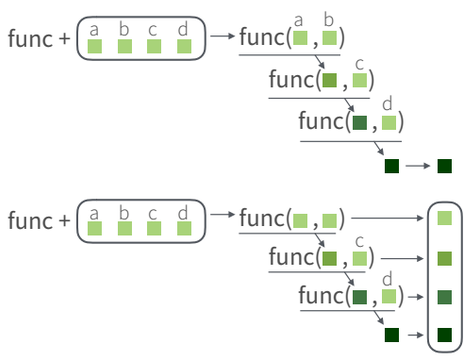


cross2(.x, .y, .filter = NULL) All combinations of .x and .y. Also **cross**, **cross3**, **cross_df**. *cross2(1:3, 4:6)*



set_names(x, nm = x) Set the names of a vector/list directly or with a function. *set_names(x, c("p", "q", "r"))* *set_names(x, tolower)*

Reduce Lists



reduce(.x, .f, ..., .init, .dir = c("forward", "backward")) Apply function recursively to each element of a list or vector. Also **reduce2**. *reduce(x, sum)*

accumulate(.x, .f, ..., .init) Reduce, but also return intermediate results. Also **accumulate2**. *accumulate(x, sum)*

Modify function behavior

compose() Compose multiple functions.

lift() Change the type of input a function takes. Also **lift_dl**, **lift_dv**, **lift_ld**, **lift_lv**, **lift_vd**, **lift_vl**.

rerun() Rerun expression n times.

negate() Negate a predicate function (a pipe friendly !)

partial() Create a version of a function that has some args preset to values.

safely() Modify func to return list of results and errors.

quietly() Modify function to return list of results, output, messages, warnings.

possibly() Modify function to return default value whenever an error occurs (instead of error).

map()

map() = lapply()

```
f <- function(x) x^2
```

```
lapply(1:3, f)
```

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

[[2]]
[1] 4

[[3]]
[1] 9
```

```
map(1:3, f)
```

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

[[2]]
[1] 4

[[3]]
[1] 9
```

```
1:3 %>% map(f)
```

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

[[2]]
[1] 4

[[3]]
[1] 9
```

map_if() and map_at()

```
f <- function(x) x^2

x <- list(1, "a", 3)

x %>% map(f)
Error in x^2 : non-numeric argument to binary operator
```

```
x %>% map_if(is.numeric, f)

[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] 9
```

```
x %>% map_at(c(1,3), f)

[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] 9
```

map() variants

Each one acts like map() but, has a specified output!

`map_int()`	`map_dbl()`	`map_lgl()`	`map_chr()`
integer vector	numeric vector	logical vector	character vector

purrr anonymous functions

Base R anonymous functions:

```
function(x) x^2
```

```
\(x) x^2
```

purrr anonymous functions: `~ .x^2`

```
1:3 %>% map(function(x) x^2)
```

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

[[2]]
[1] 4

[[3]]
[1] 9
```

```
1:3 %>% map(~ .x^2)
```

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

[[2]]
[1] 4

[[3]]
[1] 9
```

purrr anonymous functions

~ .x^2 creates function(x) x^2 (~ .^2 and ~ ..1^2 work, too)

~ .x + .y creates function(x, y) x + y (~ ..1 + ..2 works, too)

~ ..1 + ..2 + ..3 creates function(x, y, z) x + y + z

rerun()

Sometimes, you may want to rerun an expression multiple times (Monte Carlo simulation)

```
rerun(3, rnorm(3))
```

```
[[1]]
[1]  0.41829397  0.08074133 -0.01574542

[[2]]
[1]  0.8087670 -2.5226206 -0.4551611

[[3]]
[1]  0.1293123  0.6400740 -1.5151518
```