Iteration and Functionals with Base R and purrrr
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!)
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?

```java
while (condition) {
}
```

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

```java
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\textsuperscript{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 \texttt{while} than \texttt{repeat}, and it’s better to use \texttt{for} than \texttt{while} (Section 5.3.2), it’s better to use a functional than \texttt{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: \texttt{apply()}, \texttt{lapply()}, \texttt{sapply()}, \texttt{vapply()}, \texttt{integrate()}, \texttt{optim()}, etc.
Functionals

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

```r
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

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

anonymous function

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

[[2]]
[1] 4

[[3]]
[1] 9

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

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

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

[1]  1  4  9 16

Another example:

```r
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.
- `map2(x, y, f, ...)`: Apply a function to pairs of elements from two lists, vectors.
- `pmap(x, y, f, ...)`: Apply a function to pairs of elements from two lists or vectors.

Work with Lists
FILTER LISTS
- `pluck(x, , .default = NULL)`: Select an element by name or index, `pluck(x, "name")`, or its attribute with `attr_getter(pluck(x, "a"))`.
- `keep(x, , .p, ...)`: Select elements that pass a logical test. `keep(x, is.na)`.
- `discard(x, , .p, ...)`: Discard 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)`.

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, , .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)`.

RESHAPE LISTS
- `flatten(x)`: Remove a level of indexes from a list. Also `flatten_chr`, `flatten_db`, `flatten_dfc`, `flatten_int`, `flatten_lgl`, `flatten(x)`.
- `transpose(.x, .names = NULL)`: Transposes the index order in a multi-level list. `transpose(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))`.

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, ...)`: Apply function to elements that pass a test. Also `map_if`, `modify_if(x, is.numeric, + 2)`.

WORK WITH LISTS
- `array_tree(array, margin = NULL)`: Turn array into list. Also `array_tree(x, margin = 3)`.
- `cross2(x, y, .filter = NULL)`: All combinations of `x` and `y`. Also `cross`, `cross3`, `cross_dfc`, `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("a", "b", "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, .dir = c(“forward”, “backward”))`: Reduce, but also return intermediate results. Also `accumulate2`, `accumulate(x, sum)`.

Modify function behavior
- `compose()`: Compose multiple functions. `compose()`.
- `lift()`: Change the type of input a function takes. Also `lift_dlv`, `lift_dv`, `t_lif`, `lift_eval`, `lift_vl`.
- `runif()`: Rerun expression `n` times. `runif()`.
- `negate()`: Negate a predicate function (a pipe friendly!) `negate()`.
- `quietly()`: Modify function to return list of results, output, messages, warnings. `quietly()`.
- `possibly()`: Modify function to return default value whenever an error occurs (instead of error). `possibly()`.
map()

map() = lapply()

f <- function(x) x^2

<table>
<thead>
<tr>
<th>lapply(1:3, f)</th>
<th>map(1:3, f)</th>
<th>1:3 %&gt;% map(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[1]]</td>
<td>[[1]]</td>
<td>[[1]]</td>
</tr>
<tr>
<td>[[2]]</td>
<td>[[2]]</td>
<td>[[2]]</td>
</tr>
<tr>
<td>[[3]]</td>
<td>[[3]]</td>
<td>[[3]]</td>
</tr>
</tbody>
</table>
map_if() and map_at()

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

[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] 9

[[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:

\[
\text{function}(x) \ x^2
\]
\[
\backslash(x) \ x^2
\]

purrr anonymous functions:  \(~ .x^2\)

\[
1:3 \ %>% \ \text{map(function}(x) \ x^2)\]
\[
1:3 \ %>% \ \text{map(~ .x^2)}\]

[[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