Working with multiple instances

Johannes Signer

March 2023

Motivation

- Most telemetry studies deal with more than one animal.
- Dealing with multiple animals can complicate analyses significantly.
- Some packages provide an infrastructure to deal with multiple animals (e.g. `adehabitat`, `move`).
- `amt` has very limited support for dealing with multiple animals, but relies on the infrastructure provided by the `purrr` package. The hope is, that this adds additional flexibility.

Examples

- Weekly home-range size of bears.
- Does home-range size differ between different sexes, treatments?
- Individual-level habitat selection?
- What is the mean step length for different animals during day and night?

Repeating tasks in R

Three possible ways to deal with multiple instances in R:

1. Copy and paste your code and make slight adaptions (**don't do this**).
2. Use a `for`-loop
3. Use functional programming

Let's consider the following problem: We want to calculate the mean step length of each fisher from the `amt_fisher` data set:

```
library(amt)
data(amt_fisher)
unique(amt_fisher$name)
```

```
[1] "Leroy"   "Ricky T" "Lupe"    "Lucile"
```

For the first animal Leroy[1]

```
amt_fisher |> filter(name == "Leroy") |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 186.9954
```

[1] Note, I am ignoring sampling rates here.

Now we could do the same for the other three individuals:

```
amt_fisher |> filter(name == "Ricky T") |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 41.9739
```

```
amt_fisher |> filter(name == "Lupe") |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 61.04558
```

```
amt_fisher |> filter(name == "Lucile") |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 85.95821
```

- We would also need an other vector to save the results.
- While this approach might be OK for only a few individuals, it becomes very tedious for many animals or if you would like to add an additional grouping factor (say, we would also want to calculate the mean step for each day).
- `for`-loops can be useful here.

Note, only the name of the animal changes

```
amt_fisher |> filter(name == "Ricky T") |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 41.9739
```

```
amt_fisher |> filter(name == "Lupe") |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 61.04558
```

```
amt_fisher |> filter(name == "Lucile") |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 85.95821
```

We could use a variable to store the name of animal currently under evaluation, for example `i`:

```
i <- "Ricky T"
amt_fisher |> filter(name == i) |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 41.9739
```

```
i <- "Lupe"
amt_fisher |> filter(name == i) |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 61.04558
```

```
i <- "Lucile"
amt_fisher |> filter(name == i) |>
  step_lengths() |> mean(na.rm = TRUE)
```

```
[1] 85.95821
```

- `i` takes the name of the animal that is currently under evaluation. Once the code block is executed, `i` is updated to the next name.
- Everything else is the **same**.

A `for`-loop has one variable, that changes for each go.

```
for (i in c("Leroy", "Ricky T", "Lupe", "Lucile")) {
  # Do something
}
```

The body of the looped (everything between `{` and `}`) is executed **four** times. Each time the value of `i` changes.

We can use such a loop to calculate the mean step length for each animal:

```
for (i in c("Leroy", "Ricky T", "Lupe", "Lucile")) {
  amt_fisher |> filter(name == i) |>
    step_lengths() |> mean(na.rm = TRUE)
}
```

Finally, we have to take care of the results and save them in a new variable, which I called `res` here.

```
res <- c()
j <- 1
for (i in c("Leroy", "Ricky T", "Lupe", "Lucile")) {
  res[j] <- amt_fisher |> filter(name == i) |>
    step_lengths() |> mean(na.rm = TRUE)
  j <- j + 1
}
```

- `for`-loops are a significant improvement compared to approach #1.
- However, `for`-loops *can* become a bit tedious if we have multiple grouping instances. We potentially need nested `for`-loops.

A slightly deeper look at R

What is a `tibbles`?

- `tibbles` are *modern* `data.frames`.
- A tibble can have list columns.
- A list is an other data structure in R, that can hold any other data structure.

With list columns it is easy to have more complex splits of your data (e.g., animals/seasons/weeks).

What is a list?

Lists are yet an other data structures for R. Lists are can contain any object (even other lists).

```
l <- list("a", b = 1:10, x = list(list(1:10)))
str(l)
```

```
List of 3
$ : chr "a"
$ b: int [1:10] 1 2 3 4 5 6 7 8 9 10
$ x:List of 1
..$ :List of 1
.. ..$ : int [1:10] 1 2 3 4 5 6 7 8 9 10
```

Examples for lists

```
x <- list(a = 1:3, b = list(1:3))
x[[1]]
```

```
[1] 1 2 3
```

```
x$a
```

```
[1] 1 2 3
```

```
x[["b"]]
```

```
[[1]]
[1] 1 2 3
```

What is functional programming?

> *Simply put, FP is exactly what it sounds like. If you are doing something more than once, it belongs in a function. In FP, functions are the primary method with which you should carry out tasks. All actions are just (often creative) implementations of functions you've written.* towardsdata-science.com

The `apply`-family

These functions apply a function on a `matrix`, `vector` or `list`.
For matrices:

- `apply()` (we won't cover this here in more detail)

For vectors and lists

- `lapply()`
- `sapply()`

lapply()

`lapply()` applies a function to each element of a list (or a vector) and returns list, then `lapply` can be used.

```
l <- list(1:3, 2)
lapply(l, length)
```

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

[[2]]
[1] 1
```

We could achieve the same with a `for`-loop:

```
res <- list()
for (i in 1:2) {
  res[[i]] <- length(l[[i]])
}
```

Here it would make more sense to use `sapply` (R will try to simplify the data structure of the result).

```
sapply(l, length)
```

```
[1] 3 1
```

Note, we used the shortest possible way, it is also possible to explicitly work with the object the function is applied to.

```
sapply(l, function(x) length(x))
```

```
[1] 3 1
```

Since `length` only uses one argument (here `x`), we do not have to explicitly call the function.
This can be shorted (since R 4.1) to

```
sapply(l, \(x) length(x))
```

```
[1] 3 1
```

The `purrr` package provides a more type-stable way to `*apply()` functions. These are called `map_*()`.

- `lapply()` -> `map()`
- `sapply()` -> `map_*()`
  - `map_lgl()` for logical values
  - `map_dbl()` for doubles
  - `map_int()` for integers
  - `map_chr()` for text

In addition there variants of all `map*()` functions that take two inputs (`map2_*()`) and many inputs (`pmap_*()`).

```
library(purrr)
map(l, length)
```

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

[[2]]
[1] 1
```

Better

```
map_int(l, length)
```

```
[1] 3 1
```

Again, it is possible to access the object directly. This can be done as before with `function(<var>)`, `\(x)` or `~`. When using `~` the object currently under evaluation can be accessed with `.` or `.x`.

```
map_int(l, function(x) length(x))
```

```
[1] 3 1
```

```
map_int(l, ~ length(.))
```

```
[1] 3 1
```

```
map_int(l, ~ length(.x))
```

```
[1] 3 1
```

An example for `map2_*`:

```
a <- 1:4
b <- 4:1

map2_dbl(a, b, ~ .x + .y)
```

```
[1] 5 5 5 5
```

Nest and unnest

An example data set

```
set.seed(12)
dat <- data.frame(
  id = rep(1:10, each = 10),
  x = runif(100),
  y = runif(100)
)
```

We can use `nest` and `unnest` to create so called `list`-columns.

```
dat |> nest(data = c(x, y))
```

```
# A tibble: 10 x 2
      id data
   <int> <list>
 1     1 <tibble [10 x 2]>
 2     2 <tibble [10 x 2]>
 3     3 <tibble [10 x 2]>
 4     4 <tibble [10 x 2]>
 5     5 <tibble [10 x 2]>
 6     6 <tibble [10 x 2]>
 7     7 <tibble [10 x 2]>
 8     8 <tibble [10 x 2]>
 9     9 <tibble [10 x 2]>
10    10 <tibble [10 x 2]>
```

```
dat |> nest(data = -id)
```

```
# A tibble: 10 x 2
      id data
   <int> <list>
 1     1 <tibble [10 x 2]>
 2     2 <tibble [10 x 2]>
 3     3 <tibble [10 x 2]>
 4     4 <tibble [10 x 2]>
 5     5 <tibble [10 x 2]>
 6     6 <tibble [10 x 2]>
 7     7 <tibble [10 x 2]>
 8     8 <tibble [10 x 2]>
 9     9 <tibble [10 x 2]>
10    10 <tibble [10 x 2]>
```

We can then work on the nested column(s), using `mutate` in combination with `map_*`:

```
dat |> nest(data = -id) |>
  mutate(centroid.x = map_dbl(data, ~ mean(.x$x)))
```

```
# A tibble: 10 x 3
      id data               centroid.x
   <int> <list>                  <dbl>
 1     1 <tibble [10 x 2]>       0.315
 2     2 <tibble [10 x 2]>       0.444
 3     3 <tibble [10 x 2]>       0.410
 4     4 <tibble [10 x 2]>       0.672
 5     5 <tibble [10 x 2]>       0.459
 6     6 <tibble [10 x 2]>       0.562
 7     7 <tibble [10 x 2]>       0.519
 8     8 <tibble [10 x 2]>       0.575
 9     9 <tibble [10 x 2]>       0.418
10    10 <tibble [10 x 2]>       0.422
```

Lets come back to our example

First, lets use `nest()`:

```
amt_fisher |> nest(data = -name)
```

```
# A tibble: 4 x 2
  name    data
  <chr>   <list>
1 Leroy   <trck_xyt [919 x 5]>
2 Ricky T <trck_xyt [8,958 x 5]>
3 Lupe    <trck_xyt [3,004 x 5]>
4 Lucile  <trck_xyt [1,349 x 5]>
```

Now we can iterate over all animals

```
amt_fisher |> nest(data = -name) |>
  mutate(mean.sl = map_dbl(data, ~ step_lengths(.x) |>
                             mean(na.rm = TRUE)))
```

```
# A tibble: 4 x 3
  name    data                    mean.sl
  <chr>   <list>                    <dbl>
1 Leroy   <trck_xyt [919 x 5]>      187.
2 Ricky T <trck_xyt [8,958 x 5]>     42.0
3 Lupe    <trck_xyt [3,004 x 5]>     61.0
4 Lucile  <trck_xyt [1,349 x 5]>     86.0
```

The approach of list-columns makes it easy to have several grouping instances.

```
amt_fisher |> mutate(wk = lubridate::week(t_)) |>
  nest(data = -c(name, wk)) |>
  mutate(mean.sl = map_dbl(data, ~ step_lengths(.x) |>
                                   mean(na.rm = TRUE)))
```

```
# A tibble: 21 x 4
   name        wk data                    mean.sl
   <chr>    <dbl> <list>                    <dbl>
 1 Leroy        6 <trck_xyt [47 x 5]>       12.6
 2 Leroy        7 <trck_xyt [330 x 5]>     204.
 3 Leroy        8 <trck_xyt [261 x 5]>     239.
 4 Leroy        9 <trck_xyt [281 x 5]>     145.
 5 Ricky T      6 <trck_xyt [232 x 5]>      40.6
 6 Ricky T      7 <trck_xyt [737 x 5]>      40.3
 7 Ricky T      8 <trck_xyt [1,345 x 5]>    46.1
 8 Ricky T      9 <trck_xyt [1,488 x 5]>    47.1
 9 Ricky T     10 <trck_xyt [1,290 x 5]>    42.3
10 Ricky T     11 <trck_xyt [1,280 x 5]>    36.2
# ... with 11 more rows
```

Take-home messages

1. Available data structures can be used to deal with multiple animals.
2. Lists are just “containers” that contain objects.
3. List columns are a great way to organize complex data structures.