Programming in R

Sam Keat

PhD Student - UK Dementia Research Institute, Cardiff

KeatS@cardiff.ac.uk

UK Dementia Research Institute

Cardiff University
Prifysgol Caerdydd

Alzheimer's Society
United Against Dementia

Programming

The implementation of logic to facilitate specified computing operations and functionality

- Writing functions()
- Creating loops/if statements
- Creating scripts

What we will cover...

- Conditional Execution
- Defining Function Arguments
- Explicit Constraints
- Dot-dot-dot (...)
- Pipes
- Iterations with purrr
- While loops
- Other loops – purrr functions
- The map family
- Shortcuts
- Multiple arguments
- walk

Conditional Execution

- In R, the conditional execution of statements are performed within `if()` and {} blocks of code.
- To start with, code is easier to understand (by you and everyone!) if you separate the lines and use indentations. Not like this:

```
myFunction <- function(x) {if (x > 3) {return(x - 3)} else {return(x)}}
```

- Code should be easy to read!

```
myFunction <- function(x) {
  if (x > 3) {
     return(x - 3)
   } else {
     return(x)
  }
}
```

Defining Function Arguments

- There are two types of arguments: **Mandatory** and **Optional**
- The mandatory arguments are **always at the beginning** of the list of arguments, followed by **optional arguments** and their **default values**.
- Example:

```
pow <- function(x, y = 2) {
   return(x ** y)
}
```

- On your R console, type `` `pow(3)` `` and `` `pow(3,3)` `` and see what the function does!

Defining Function Arguments

- There are two ways of passing the values to a function: **by order** and **by name.**
- Check the description of mean() by typing `?mean`

```
mean(1:101, ,TRUE)
```

- This is **bad programming! X** because it means you have to remember the command and each of its arguments (try doing that for 1000 functions!).
- Passing them by name means you can change the order:

```
mean(na.rm = TRUE, x = 1:101)
```

- Best practice is to define include the first argument (can be without name) of the function as the input data to be processed:

```
mean(1:101, na.rm = TRUE)
```

Defining Function Arguments

- Adding restrictions to the function means they can be more efficient
- For example:

```
midValue <- function(x) {
  if (length(x) %% 2 == 0) {
    stop("'x' has an even number of elements", call. = FALSE) }
midIndex <- (length(x) + 1) / 2
  return (x[midIndex])
}
```

- The stop function is executed when the modulus (remainder from division) is zero. A good error checking mechanism (even gives a message!)

Defining Function Arguments

- The previous code can be simplified by using: `stopifnot()`

```
midValue <- function(x) {
  stopifnot(length(x) %% 2 == 1)
  midIndex <- (length(x) + 1) / 2
  return (x[midIndex])
}
```

- However we get a generic error message, not as meaningful as previous one. What kind of error message do we get when we write more than one condition in `stopifnot()`?

```
calMean = function(x) {
  stopifnot( exprs = {
    mean(x) == 4
    length(x) == 4
  })
}
```

Dot-Dot-Dot (...)

- In coding, this is called an **ellipsis**.
- An elipsis means that the function can take any number of named or unnamed arguments

```
print(x, ...)
```

- For example: We can use ... to pass those additional arguments on to another function. Essentially, placeholders for other arguments.

```
i01 <- function(y, z) {
    list(y = y, z = z)
}
i02 <- function(x, ...) {
    i01(...)
}
```

Dot-Dot-Dot (...)

```r
str(i02(x = 1, y = 2, z = 3))
```

- Output:

```
#> List of 2
#> $ y: num 2
#> $ z: num 3
```

- By adding numbers at the end, it is possible to refer to elements of ... by position (what position the generic arguments will sit in).

```r
i03 <- function(...) {
    list(first = ..1, third = ..3)
}

str(i03(1, 2, 3))
```

Dot-Dot-Dot (...)

- Output:

```
#> List of 2
#> $ first: num 1
#> $ third: num 3
```

- More useful is `list(...)`, which evaluates the arguments and stores them in a list. Very useful when working with data!

```
i04 <- function(...) {
    list(...)
}

str(i04(a = 1, b = 2))
```

Pipes %>%

- There are two types of pipeable functions: **transformations** and **side-effects**.
- **Transformations** are where an object is passed to the function's first argument and a modified object is returned.
- With **side-effects**, the passed object is not transformed. Instead, the function performs a function on that object, such as drawing a plot or saving a file.

```
print_missings <- function(df) {
    n <- sum(is.na(df))
    cat("Missing values: ", n, "\n", sep = "")
    invisible(df)
}
```

Pipes %>%

- If we use our newly created `print_missings()` function, the `invisible()` command means that the input data frame will not get printed out (echo=false) but we can still use it in a pipe.

```
library(tidyverse)
diamonds %>%
    print_missings() %>%
    mutate(carat = ifelse(carat < 0.25, NA, carat)) %>%
    print_missings()
```

```
## Missing values: 0
## Missing values: 573
```

Iterations with purrr

- We want to keep code **efficient** and **less repetitive**: performing the same thing on multiple inputs, repeating the operation on multiple columns, or on different datasets.
- To help achieve this, **iterations** are used. For example:

```
library(tidyverse)
```

```
rescale <- function(x) {
    y <- min(x, na.rm = TRUE)
    return((x - y) / (max(x, na.rm = TRUE) - y))
}
```

```
df <- data.frame(a = rnorm(10), b = rnorm(10), c = rnorm(10), d = rnorm(10))
df$a <- rescale(df$a)
df$b <- rescale(df$b)
df$c <- rescale(df$c)
df$d <- rescale(df$d)
```

Iterations with purrr

- We can simplify the code with an easy for loop:

```r
for (i in seq_along(df)) {
    df[[i]] <- rescale(df[[i]])
}
```

- For example:

```r
means <- c(0, 1, 2)
out <- vector("list", length(means))
for (i in seq_along(means)) {
    n <- sample(10, 1)
    out[[i]] <- rnorm(n, means[[i]])
}

str(out)
```

While loops

- While loops work differently to for loops.

```
while (condition) {
    # body
}
```

- The while loop works in the background until a condition is met, whilst being more general than a for loop. Any for loop can be rewritten as a while loop but not any while loop can be rewritten as a for loop.

```
for (i in seq_along(x)) {
    # body
}
i <- 1
while (i <= length(x)) {
    # body
    i <- i + 1
}
```

While loops

- **Example:** the number of times we need to flip a coin to get three heads in a row:

```
flip_coin <- function() {
    sample(c("T", "H"), 1)
}
numFlips <- 0
numHeads <- 0
while (numHeads < 3) {
    if (flip_coin() == "H") {
        numHeads <- numHeads + 1
    } else {
        numHeads <- 0
    }
    numFlips <- numFlips + 1
}

numFlips
```

Other loops - purr functions

- **purrr** is a package that helps to enhance R's functional programming toolkit
- **purrr** functions help to break common challenges in list manipulation into independent pieces.
- Base R has family of functions known as **"apply family"**, that eliminates the need for many common for loops, `apply(), lapply(), tapply()`
- **purrr** has a family of functions called the "**map family**".
- Each function takes a vector as input, applies a function to each piece, and then returns a new vector that has the same length as the input.

The `map` family

- Essentially, `map()` is the **tidyverse** equivalent of the base R `apply` family of functions.
- The basic syntax is `map(.x, .f, ...)` where:
  - `.x` is a list, vector or dataframe
  - `.f` is a function
  - `map()` will then apply `.f` to each element of `.x` in turn.

The `map` family

- We can use the `map` function to compute the mean, median and standard deviation of previous dataset.

```
map_dbl(df, mean)
map_dbl(df, median)
map_dbl(df, sd)
```

- Go to the help page of `map_dbl()`, you can see that we find again the special arguments `...` , meaning that we can pass the arguments to the selected function.

```
map_dbl(df, mean, na.rm = TRUE)
```

The `map` family

- We can even use a string or a position (integer) to extract components from the input data - very useful when working with big datasets!

```
x <- list(x=list(a=1, b=2, c=3), y=list(a=4, b=5, c=6), z=list(a=7, b=8,
c=9))

x %>% map_dbl("a")
x y z
1 4 7

x %>% map_dbl(2)

x y z
2 5 8
```

`map_` functions

- One property of the `map()` function is that **it will always return a list**.
- To change the output data type, we can use multiple versions of `map_*():`
  - `map_lgl()` returns a logical.
  - `map_int()` returns a integer vector.
  - `map_dbl()` returns a double vector.
  - `map_chr()` returns a character vector.
  - `map_df()` returns a data frame.

Shortcuts

- Here are a few shortcuts to save typing!
- Fit a linear model to each group in a dataset. Theis example splits up the mtcars dataset into three pieces and fits the linear model to each piece.

```
models <- mtcars %>%
    split(.$cyl) %>%
    map(function(df) {lm(mpg ~ wt, data = df)})
```

- We replace the anonymous function with **purrr's** shortcut

```
models <- mtcars %>%
    split(.$cyl) %>%
    map(~lm(mpg ~ wt, data = .)})
```

Shortcuts

- Notice the "." we included, this is a placeholder for the dataset we've piped in (in this case, the "mtcars" dataset) so we can access parts of it ($)
- When we are looking at many models, we might want to extract the summary statistic, such as the R^2 value. We can use `summary()` and then extract the component called `r.squared`.

```
models <- mtcars %>%
    map(summary) %>%
    map_dbl(~.$r.squared)
```

Multiple Arguments

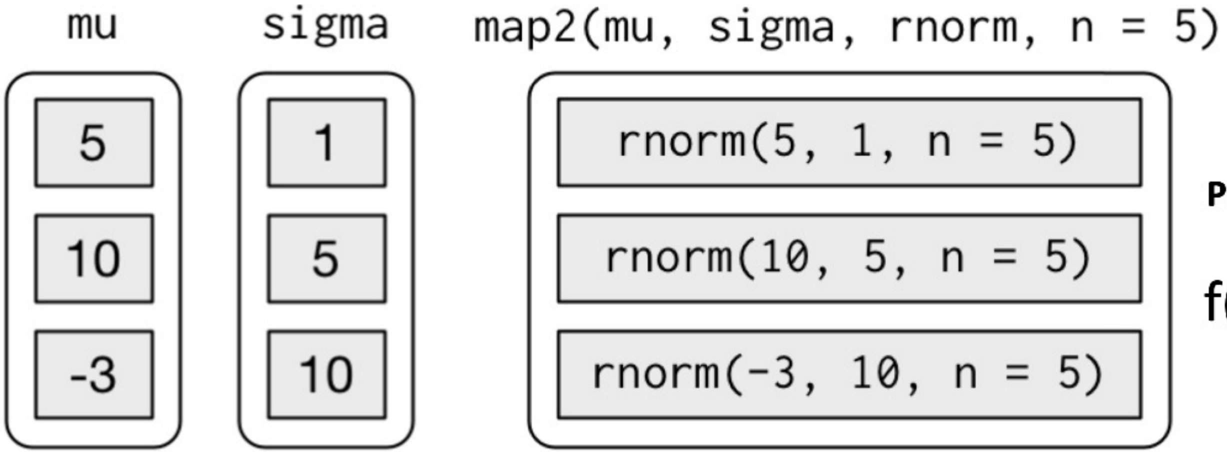
- **purrr** gives us the option to include more than one input in parallel with `map2()` and `pmap()`.
- Imagine we would like to simulate some random normal distributions with different means, and each vary, we could do:

```
mu <- list(5, 10, -3)
sigma <- list(1, 5, 10)
map2(mu, sigma, rnorm, n = 5) %>%
   str()
```

```
## List of 3
## $ : num [1:5] 5.15 4.93 4.56 5.85 6.12
## $ : num [1:5] 11.69 14.03 5.83 3.21 13.83
## $ : num [1:5] 12.5 -4.78 4.57 13.17 -2.11
```

Multiple Arguments

- The code can be understood by the following figure:



Probability Density Function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

Multiple Arguments

- Further to map2, you should also have map3 (for 3 inputs), and map4 (for 4), etc. For this purpose, **purrr** has the function `pmap()`.

```
n <- list(1, 3, 5)
arguments <- list(n, mu, sigma)
arguments %>%
    pmap(rnorm) %>%
    str()
```

- We can go even further by increasing the complexity of the problem using the `invoke_map()` function.

```
funcs <- c("runif", "rnorm", "rpois")
params <- list(
    list(min = -1, max = 1),
    list(sd = 5),
    list(lambda = 10)
)
invoke_map(funcs, params, n = 5) %>%
    str()
```

walk

- `walk` is an alternative to map that we use we call a function for its side effects, disregarding its return value.

```
x <- list(1, "a", 3)
x %>%
    walk(print)
```

```
[1] 1
[1] "a”
[1] 3
```

- Really useful when outputting datasets in lists! (such as microarray data)
- Similar to `map()`, **purrr** also has `walk2()` and `pwalk()`

walk2

```
library(tidyverse)
df0 <- tibble(x = 1:3, y = rnorm(3))
df1 <- tibble(x = 1:3, y = rnorm(3))
df2 <- tibble(x = 1:3, y = rnorm(3))
animalFrames <- tibble(animals = c('sheep', 'cow', 'horse'), frames =
list(df0, df1, df2))

animalFrames %>%
    walk2(
        .x = .$animals,
        .y = .$frames,
        .f = ~ write_csv(.y, str_c("test_", .x, ".csv"))
    )
```

pwalk

- pmap() and pwalk() allow you to provide any number of arguments in a list.
- Syntax:

  pwalk(.l, .f, ...)

```
ds_mt <-
mtcars %>%
    rownames_to_column("model") %>%
    mutate(am = factor(am, labels = c("auto", "manual"))) %>%
    select(model, mpg, wt, cyl, am) %>%
    sample_n(3)
foo <- function(model, am, mpg){
    print( paste("The", model, "has a", am, "transmission and gets",
    mpg, "mpgs.") )
}
```

pwalk

```
ds_mt %>%
    select(model, am, mpg) %>%
    pwalk(
        .l = .,
        .f = foo
    )
```

```
[1] "The Toyota Corona has a auto transmission and gets 21.5 mpgs.”
[1] "The Cadillac Fleetwood has a auto transmission and gets 10.4 mpgs.”
[1] "The Merc 280C has a auto transmission and gets 17.8 mpgs."
```

Interactive R workshop!