Fun with **Fun**ctions

R Ladies February 2020

Kaylea Haynes - @kayleahaynes, kaylea.haynes@peak.ai

```
goal_function <- function(skill_level){

  if (skill_level == "beginner"){

    ideal_outcome <- "understand what functions are and start using functions day to day"

  } else if (skill_level == "intermediate"){

    ideal_outcome <- "learn more advanced techniques for using functions in your workflow"

  } else if (skill_level == "advanced") {

    ideal_outcome <- "feel warm and fuzzy for teaching sharing your wisdom with your peers"

  }

  outcome <- paste(ideal_outcome, "and/or teach me something I didn't know about functions
in R")

  return(outcome)

}
```

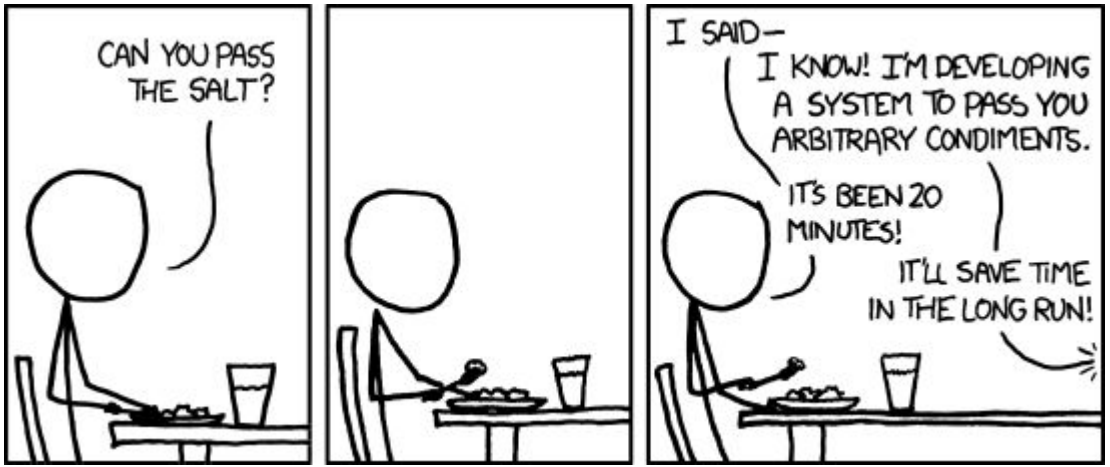
Plan

1. Dive straight in to writing some example functions
    a. Step by step guide at building up functions from the ground up
    b. Have some time to practise some examples
2. Take a step back and explore good coding practices for functions
3. Take your functions a step further
    a. Apply, lapply, mapply etc
    b. Pipes
4. Questions/Discussion

An introduction to functions

Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting

1. Reduces the chance of error
2. If something changes you only need to change your code in one place
3. Can name and document your functions to make your code easier to read and understand
4. Can be easier for collaboration



You should consider writing a function whenever you've copied and pasted a block of code more than twice

Functions are created using `function()`

Analyse your code to work out what your function inputs should be. These can be rewritten as temporary variables

```
print("Hello Ellen, welcome to R Ladies!")
print("Hello Bethan, welcome to R Ladies!")
print("Hello Naomi, welcome to R Ladies!")
```

```
my_func <- function(arguments){
  # do some stuff
  return(function_result)
}
```

```
r_lady <- "Bethan"

print(paste("Hello", r_lady, "welcome to R Ladies!"))
```

Steps to create a function:

1. Think of a good name
2. List the arguments (function inputs) inside function.
3. Put your code within the function body.
4. Decide what you want the function to return

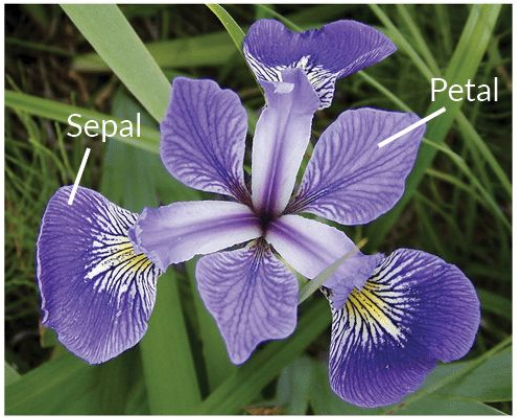
Note - It's easier to start with working code and turn it into a function

```
welcome_r_ladies <- function(r_lady){

    return(paste("Hello", r_lady,
"welcome to R Ladies!"))
}


welcome_r_ladies("Ellen")
```

If we decide to make a change to the code we only need to do it in one place

```
print("Hello Ellen, welcome to R
Ladies, your birthday is on the 10th of
August")
print("Hello Bethan, welcome to R
Ladies, your birthday is on the 20th of
July")
print("Hello Naomi, welcome to R
Ladies, your birthday is on the 30th of
May")


welcome_r_ladies <- function(r_lady,
dob){
  return(paste("Hello", r_lady,
"welcome to R Ladies, your birthday is
on the", format(as.Date(dob), '%d -
%B')))
}


welcome_r_ladies("Ellen", "1990-08-01")
```

Let’s go through an example



Iris Versicolor

Iris Setosa

Iris Virginica

There are a number of datasets that can be called directly in R from the datasets package. For the following example we will use the iris data set.

```
head(datasets::iris, 3)
```

```
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 5.1          3.5         1.4          0.2         setosa
 4.9          3.0         1.4          0.2         setosa
 4.7          3.2         1.3          0.2         setosa
```

Function to normalise columns

For each numeric column of the iris data set I want to normalise so that the values are between 0 and 1.

There's a lot of copy and pasted code and it could have been easy to make a mistake.

```
data <- iris

data$Sepal.Length <- (data$Sepal.Length -
min(data$Sepal.Length, na.rm = TRUE)) /
(max(data$Sepal.Length, na.rm = TRUE) -
min(data$Sepal.Length, na.rm = TRUE))

data$Sepal.Width <- (data$Sepal.Width -
min(data$Sepal.Width, na.rm = TRUE))
/(max(data$Sepal.Width, na.rm = TRUE) -
min(data$Sepal.Width, na.rm = TRUE))

data$Petal.Length <- (data$Petal.Length -
min(data$Petal.Length, na.rm = TRUE))
/(max(data$Petal.Length, na.rm = TRUE) -
min(data$Petal.Length, na.rm = TRUE))

data$Petal.Width <- (data$Petal.Width -
min(data$Petal.Width, na.rm = TRUE))
/(max(data$Petal.Width, na.rm = TRUE) -
min(data$Petal.Width, na.rm = TRUE))
```

Generalise the code

We can see that each line of code is the same apart from the call to the column name of interest.

Our first step is to rewrite the code by creating a temporary variable.

There's still a lot of copy and pasting!

```
x <- data$Sepal.Length

(x - min(x, na.rm = TRUE)) / (max(x,
na.rm = TRUE) - min(x, na.rm = TRUE))

x <- data$Sepal.Width

(x - min(x, na.rm = TRUE)) / (max(x,
na.rm = TRUE) - min(x, na.rm = TRUE))

x <- data$Petal.Length

(x - min(x, na.rm = TRUE)) / (max(x,
na.rm = TRUE) - min(x, na.rm = TRUE))

x <- data$Petal.Width

(x - min(x, na.rm = TRUE)) / (max(x,
na.rm = TRUE) - min(x, na.rm = TRUE))
```

Create a function

The function normalises numeric data so that the values are between 0 and 1. Thus we call our function “normalise”.

```
normalise <- function(x){

x_normalised <- (x - min(x, na.rm =
TRUE)) / (max(x, na.rm = TRUE) - min(x,
na.rm = TRUE))

return(x_normalised)

}

normalise(data$Sepal.Length)

normalise(data$Sepal.Width)

normalise(data$Petal.Length)

normalise(data$Petal.Width)
```

Set function output to a variable

It would possibly be more useful to set the output of the function as its own variable so it can be used later in the code.

Some options:

- Replace the original data. This has the benefit that you won't use the raw data later in the code when you wanted to use the normalised data.
- Create a new variable for your output. This is useful when you want to use the raw data elsewhere in the code.
- If using a data frame add another column for your output.

```
data$Sepal.Length <- normalise(data$Sepal.Length)

data$Sepal.Width <- normalise(data$Sepal.Width)

data$Petal.Length <- normalise(data$Petal.Length)

data$Petal.Width <- normalise(data$Petal.Width)

                              ---

data$Sepal.Length_norm <- normalise(data$Sepal.Length)

data$Sepal.Width_norm <- normalise(data$Sepal.Width)

data$Petal.Length_norm <- normalise(data$Petal.Length)

data$Petal.Width_norm <- normalise(data$Petal.Width)
```

Loop over all columns

Sometimes you might want to apply your function to all columns of a data frame. Note we will look at more advanced ways of doing this later.

There are different ways to save the output of your function depending on your application. A few options are:

- Replace the original column
- Create a new data set
  - If you know the dimensions of your dataset you can create an empty data set
  - If you don't know the dimensions create and empty object and then append on each loop

```
for (i in 1:4){
  data[,i] <- normalise(data[,i])
}
```

```
new_data <- matrix(nrow = nrow(data),
ncol = ncol(data)-1)
for (i in 1:4){
  new_data[,i] <- normalise(data[,i])
}
```

```
new_data <- NULL
for (i in 1:4){
  new_data <- cbind(new_data,
normalise(data[,i]))
}
```

Return value

The value returned by the function is usually the last statement it evaluates.

You can return early using `return()` this is useful if you want to return conditionally.

Note: I've got into the habit of explicitly stating return() even if it is the last statement in my script.

```
welcome_r_ladies <- function(r_lady){

     if (r_lady %in% c("Hello", "Hello")){
return(paste("Hello", r_lady, "thanks for
organising a great session for us!"))
}
     paste("Hello", r_lady, "welcome to R
Ladies!")
}
```

```
welcome_r_ladies <- function(r_lady){

     if (r_lady %in% c("Hello", "Hello")){
return(paste("Hello", r_lady, "thanks for
organising a great session for us!"))
}
     return(paste("Hello", r_lady, "welcome to
R Ladies!"))
     }
```

Returning multiple values

Sometimes you might want to return multiple values. This is useful if you want different outputs using similar code and to avoid repetition you could just do it at the same time.

To return multiple objects use a list.

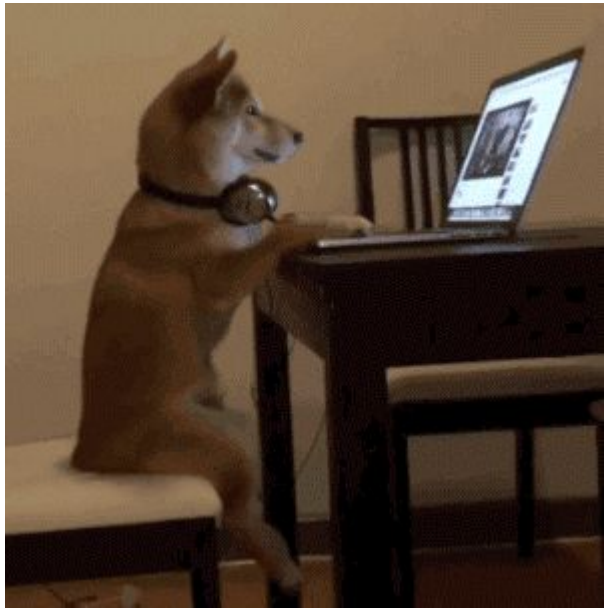
```
mean_and_sd <- function(x){
  x_mean <- mean(x)
  x_sd <- sd(x)
  return(list(x_mean, x_sd))
}

output <- mean_and_sd(datasets::iris[,1])
output[[1]]
output[[2]]
```

```
mean_and_sd <- function(x){
  x_mean <- mean(x)
  x_sd <- sd(x)
  return(list(mean1 = x_mean, sd1 = x_sd))
}

output <- mean_and_sd(datasets::iris[,1])
output$mean1
output$sd1
```

Your turn

Try creating your own functions.

- Think about some code you copy and paste a lot.
- Try and generalise the code to use `x`
- What is a sensible name for your function?
- What would the inputs to your function be?
- What are your return values?



Alternatively choose one of Rs data sets (*mtcars, faithful* etc) write a function to return the minimum and maximum values of each of the columns.

Good practice

Function names

Function names should be short but make it clear what the function does.

Ideally function names should be verbs and the arguments nouns.

If function names are multiple words it's common to use snake_case. camelCase is a popular alternative. Choose the one you'd prefer. Just keep it consistent.

If you have multiple functions doing similar things it's a good idea to use a common prefix in the function names.

Arguments

Arguments normally specify the **data** to use within a function and control the **details** of the functions computation.

mean(): the data is x and the details are how much data to trim from the ends (trim) and how to handle missing values (na.rm).

In t.test(), the data are x and y, and the details of the test are alternative, mu, paired, var.equal, and conf.level.

Normally data arguments come first and detail arguments after.

Argument names

Name your arguments sensibly and logically. Your future self and peers will thank you!

Some common short names:

- `x`, `y`, `z`: vectors.
- `w`: a vector of weights.
- `df`: a data frame.
- `i`, `j`: numeric indices (typically rows and columns).
- `n`: length, or number of rows.
- `p`: number of columns.

Special Arguments

- Function names can be passed as arguments.

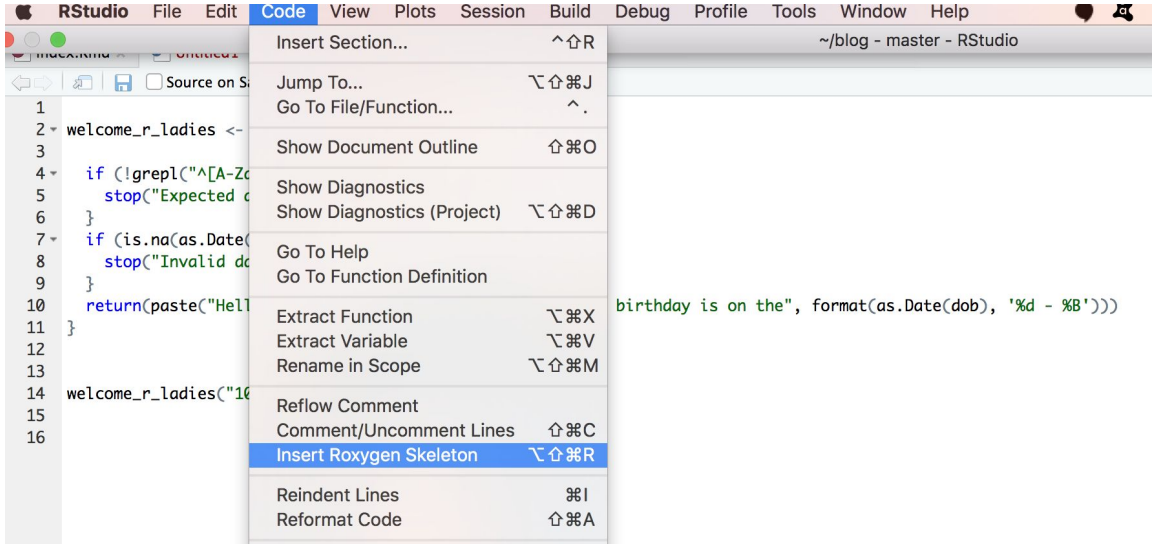
```
mean_and_sd <- function(x, func1){
     x_return <- func1(x)
}

mean1 <- mean_and_sd(iris$Sepal.Length, mean)
```

- dot-dot-dot (...) i.e. `x <- function(x, …)`
  - an arbitrary number and variety of arguments
  - passing arguments on to other functions
  - Note: A misspelled argument will not raise an error. This makes it easy for typos to go unnoticed.

Documentation - yes it's boring but just do it! You'll thank yourself later.

For each function include a short description explaining why you wrote the function and what it needs to be able to run.

R has a package roxygen which makes documenting functions super easy.



Documentation - yes it's boring but just do it! You'll thank yourself later.

R has a package roxygen which makes documenting functions super easy.

```r
#' Title
#'
#' @param r_lady
#' @param dob
#'
#' @return
#' @export
#'
#' @examples
welcome_r_ladies <- function(r_lady, dob){

  if (!grepl("^[A-Za-z]+$", r_lady)){
    stop("Expected a character string for R Ladies")
  }
  if (is.na(as.Date(dob))){
    stop("Invalid date")
  }
  return(paste("Hello", r_lady, "welcome to R Ladies, your birthday is on the", format(as.Date(dob), '%d - %B')))
}
```

- what the function does
- what are the inputs
- what is the output
- give an example which can be used to try the function

It's useful to put some some tests in your function to through as error if the output isn't as expected.

This helps others (and your future self) use your function as it was intended.

```
welcome_r_ladies <- function(r_lady,
dob){

if (!grepl("^[A-Za-z]+$", r_lady)){
stop("Expected a character string for R
Ladies")}

if (is.na(as.Date(dob))){stop("Invalid
date")}

return(paste("Hello", r_lady, "welcome
to R Ladies, your birthday is on the",
format(as.Date(dob), '%d - %B')))

}


welcome_r_ladies("Ellen", "1990-08-01")
```

Your turn:

Explore your favourite functions.

What does the function name refer to?

- What are the data arguments and what are the detail arguments ?
- What does it return?
- What are the documents like?

Look at the function(s) you wrote earlier

- Is the name sensible?
- Are the argument names sensible?
- Try and add some documentation using roxygen.

Lots more fun

Functionals (lapply() , sapply(), vapply(), mapply(), rapply(), and tapply())

A function that takes a function as an input and returns a vector as output

- Functions are used as an alternative to loops
- Loops convey that you are iterating but it isn't clear what you plan to do with the output of a for loop.
    - Functionals are easy to recognise why it is being used.

```
apply(variable, margin, function)
```

–**variable** is the variable you want to apply the function to.

–**margin** specifies if you want to apply by row (margin = 1), by column (margin = 2), or for each element (margin = 1:2). Margin can be even greater than 2, if we work with variables of dimension greater than two.

–**function** is the function you want to apply to the elements of your variable.

Functionals (lapply() , sapply(), vapply(), mapply(), rapply(), and tapply())

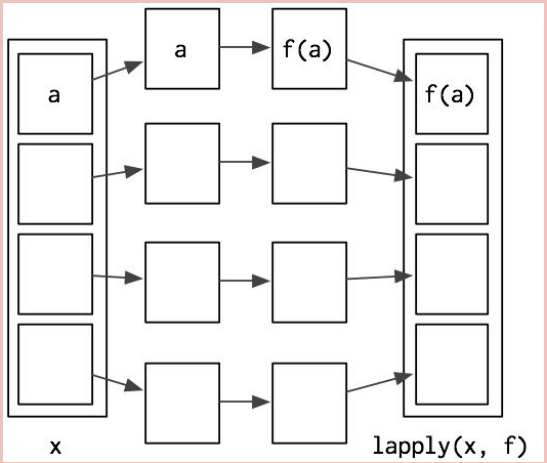
- lapply(): used for lists, vectors or data frames. Returns a list.
- sapply(): similar to lapply but it tries to guess the output. vapply() is similar to sapply() but it takes an additional argument specifying the output type. It’s safer to use vapply than sapply.
- mapply() and Map(): take multiple arguments that can change in the function.

Functionals (lapply() , sapply(), vapply(), mapply(), rapply(), and tapply())

A common use of functionals is as an alternative to for loops.

lapply() is written in C for performance in R it is essentially:

```
lapply2 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
 }
```




```
data <- iris
out <- vector("list", length(data[,1:4]))
for (i in seq_along(data[,1:4])) {
  out[[i]] <- normalise(data[[i]])
}


out <- lapply(data[,1:4], normalise)
```

Alternatively use purrr::map().

purrr::map() is the tidyverse equivalent to lapply()

There are 23 primary variants of map()

Task: explore and understand the use cases of the different map() variations.

```
purrr::map(data[,1:4], normalise)
```

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

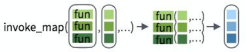
Apply Functions

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

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

~ ..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)*

RStudio

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)*

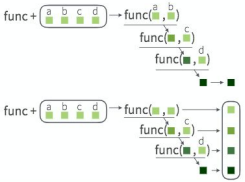
WORK WITH LISTS

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.

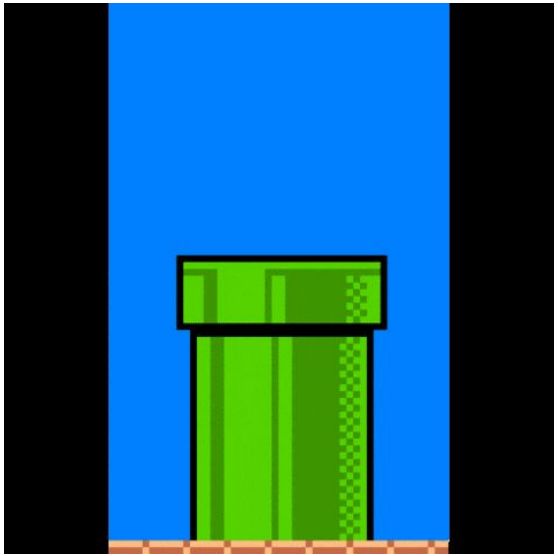
quietly() Modify function to return list of results, output, messages, warnings.

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

https://github.com/rstudio/cheatsheets/blob/master/purrr.pdf

Pipeable functions (%>%)

If you want to write your own pipeable functions, it's important to think about the return value



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

normalise_all <- function(x){
  data.frame(sapply(x, normalise))
}

mean_and_sd_all <- function(x){
  data.frame(sapply(x, mean_and_sd))
}

data[,1:4] %>%
  normalise_all() %>%
  mean_and_sd_all()
```

Using your functions in practice

There are a few options to use your functions in practise.

1. When you first write a function you could put it in the script where you plan to call it from. If you want to use it elsewhere you'll need to copy and paste!
2. If you want to use your function in multiple scripts can have a script that you source at the beginning of your script (`source('~/.function_file.R')`
3. If you want to use your functions a lot across multiple scripts/projects then build a package…

Discussion/questions