3. Base R - Functions, Functionals and the R Pipe

CT5102 - J. Duggan (University of Galway)

Functions

Functions are the natural unit of programming in the small: creating software that answers questions of immediate interest and that captures specific ideas in extending R.
— John Chambers

Overview

- Functions are building blocks in R, and are small units that can take an input, process it in some useful way, and return a result.
- Learning outcomes:
  - How to write your own function, call it via arguments, and return values based on the last evaluated expression.
  - The different ways to pass arguments to functions.
  - How to add robustness checking to your functions to minimise the chance of processing errors.
  - What an environment is, and the environment hierarchy within R.
  - How a function contains a reference to its enclosing environment and how this is used to access variables in the global environment.
  - What a functional is, namely, a function that can take another function as an argument.
  - The functional `lapply()`, and how this can be used, as an alternative to loops, to iterate over lists and vectors.
  - R's native pipe operator, and how this can be used to streamline a sequence of data processing operations.
  - How to solve all three test exercises.

Functions

- A function can be defined a group of instructions that: takes input, uses the input to compute other value, and returns a result.
- Functions are declared using the `function` reserved word, and are objects, which means they can also be passed as arguments to other functions. The general form of a function in R is:

```
function (arguments) expression
```

where:

- `arguments` provides the arguments (inputs) to a function, and are separated by commas
- `expression` is any legal R expression, and is the function body, and is usually enclosed in curly brackets (when there is more than one expression)
- the last evaluated expression is returned by the function, although the function `return()` can be also used to return values.

A First Function

- Our first function will take in a vector of numbers, and return only those that are even.
- To do this, R's modulus operator %% is used, as this returns the remainder of two numbers, following their division.
- The following snippet shows the results of using the modulus operator, as it divides the vector `1:5` by 2 and calculates the remainder.

```r
v <- 1:5
x <- v %% 2
x
#> [1] 1 0 1 0 1
```

Filtering the vector

- We first explore the logic needed to filter the input vector.
- Thinking of how the modulus function works, in this case, if it returns a remainder of 0 then we know the number is divisible by two.
- We can use this to create a logical vector that can then be used to filter the modulus result, as shown in the following code.

```
x                    # The results of v %% 2
#> [1] 1 0 1 0 1
lv <- x == 0         # Logical vector for even values
lv                   # Show the logical vector
#> [1] FALSE  TRUE FALSE  TRUE FALSE
v[lv]                # Filter the original vector
#> [1] 2 4
```

Embedding within a function

- This logic can now be embedded within an R function which we will call `evens()`
- This which takes in one argument (the original vector), and returns a filtered version of the vector that only includes even numbers.
- We will take a parsimonious approach to code writing, and just limit the function to one line of code.

```
evens <- function(v){
  v[v%%2==0]
}
x1 <- 1:7
evens(x1)
#> [1] 2 4 6
```

A Second Function - Removing Duplicates from a vector

- Our second function takes an approach of building on the work of other developer, and using an existing base R function to create a new one.
- This function will take in a vector of random numbers, and remove any duplicates.
- To remove the duplicates, we will make use of the R function `duplicated()`, which returns a logical vector that contains `TRUE` if a value is duplicated. Here is an example of its use.

```
set.seed(100)
v <- sample(1:6,7,replace = TRUE)
v
#> [1] 2 6 3 1 2 6 4
duplicated(v)
#> [1] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
```

Using duplicated() to find unique values

In order to find the set of values that are unique, we can use the information returned by duplicated(), as follows.

```
v
#> [1] 2 6 3 1 2 6 4
v[!duplicated(v)]
#> [1] 2 6 3 1 4
```

- The challenge now is to embed this logic into a function, so that it can be called as needed.
- We will call the function my_unique() that takes in a vector (one argument), and returns the unique values from the vector.
- It is also useful to write the function into a source file, let's assume the file is called my_functions.R.
- This function could just be written in one line of code, but we will break it down into a number of separate steps just to clarify the process.

my_unique() function - version 1

```
my_unique <- function(x){
  # Use duplicated() to create a logical vector
  dup_logi <- duplicated(x)
  # Invert the logical vector so that those
  # not duplicated are set to TRUE
  unique_logi <- !dup_logi
  # Subset x to store those values are unique
  ans <- x[unique_logi]
  # Evaluate the variable ans so that it is returned
  ans
}
```

Loading the function

- To load the function into R, call the source function (this is easy to do within R Studio by clicking the "Source" button).
- The function is then loaded into the workspace, and this can be confirmed by calling the `ls()` function, which returns return a vector of character strings giving the names of the objects in the specified environment.

```
source("my_functions.R")

ls()
#> [1] "evens"      "lv"          "my_unique" "v"          "x"
```

Calling `my_unique()`

Once a function is loaded in the global environment, it can then be accessed by a call. The code below shows the call, and confirms that the answer is stored in a new variable. The complete code is shown, including the vector `v`.

```
set.seed(100)
v <- sample(1:6,7,replace = T)
ans <- my_unique(v)
ans
#> [1] 2 6 3 1 4
```

Normally, when writing in R, programmers tend to reduce redundancy in the code, so that following shorter function would suffice for `my_unique()`.

```
my_unique <- function(x){
  x[!duplicated(x)]
}
```

Functions are objects

- Interestingly in R, functions are also objects, so they can be passed to functions as arguments
- *Functionals* are functions that accept functions as parameters.
- To send a function as a parameter, all that required is the function name. .

```
my_summary <- function(v, fn){
  fn(v)
}

# Call my_summary() to get the minimum value
my_summary(1:10,min)
#> [1] 1
# Call my_summary() to get the maximum value
my_summary(1:10,max)
#> [1] 10
```

Another example

```r
my_min<- function(v){
  min(v)
}
```

You can write your own functions that can be passed into another function.

```r
# Call my_summary() to get the minimum value
my_summary(1:10,my_min)
#> [1] 1
```

An anonymous function

- Furthermore, you could also write the logic of `my_min` as an *anonymous function* (i.e. it is not assigned to a variable, and so does not appear in the global environment)
- Right now this might seem like an odd thing to do, however, anonymous functions are key idea used when we start to explore functionals such as `lapply()`, and later `purrrr::map()`, to iterate over list structures, and apply an action to each list element.

```
my_summary(1:10,function(y)min(y))
#> [1] 1

my_summary(1:10,function(y)max(y))
#> [1] 10
```

Passing arguments to functions

- When programming in R, it is useful to distinguish between the *formal arguments*, which are the property of the function itself, and the actual arguments, which can vary when the function is called Wickham (2019).
- For example, the function 'sum()‘ could be called with different arguments, as shown below.

```
v <- c(1,2,3,NA)
sum(v)
#> [1] NA
sum(v,na.rm=TRUE)
#> [1] 6
```

Passing arguments

- Each function in R is defined with a set of formal arguments that have a fixed positional order, and often that is the way arguments are then passed into functions (e.g. by position).
- However, arguments can also be passed in by *complete name* or *partial name*, and arguments can also have default values.
- We can explore this via the following example for the function `f`, which has three formal arguments: `abc`, `bcd` and `bce`, and simply returns an atomic vector showing the function inputs (argument one, argument two and argument three).

```
f <- function(abc,bcd,bce){
  c(FirstArg=abc,SecondArg=bcd,ThirdArg=bce)
}
```

Passing arguments - by position

- *By position*, where the arguments are copied to the corresponding argument, which is the most common method in most programming languages. Here 1 is copied to `abc`, 2 is copied to `bcd` and 3 is copied to `bce`.

```
f(1,2,3)
#>  FirstArg SecondArg  ThirdArg
#>         1         2         3
```

Passing arguments - by complete name

- *By complete name*, where arguments are first copied to their corresponding name, before other arguments are then copied via their positions.
- The advantage of this is that the programmer calling the function does not need to know the exact position of an argument to call it, and we have seen this already with the use of the argument `na.rm` in R base functions such as `sum()`.

```
f(2,3,abc=1)
#>  FirstArg SecondArg  ThirdArg
#>         1         2         3
```

Passing arguments - by partial name

- *By partial name*, where argument names are matched, and where a unique match is found, that argument will be selected.
- A observation here is that if there is more than one match, the function call will fail. Furthermore, using partial matching can lead to confusion for someone trying to understand the code.

```
f(2,a=1,3)
#>  FirstArg SecondArg  ThirdArg
#>         1         2         3
```

Provide default values to arguments

- A very useful feature with defining arguments is that they can be allocated default values, which provides flexibility in that not all the arguments need to be called each time the function is invoked.
- We can modify the function `f` so that each argument has a arbitrary default value.

```
f <- function(abc=1,bcd=2,bce=3){
  c(FirstArg=abc,SecondArg=bcd,ThirdArg=bce)
}
```

Further flexibility in calls

Following this, the function can be called in four different ways: with no arguments, and with one, two or three arguments. In this example, a mixture of positional and complete naming matching are used.

```
f()
#>  FirstArg SecondArg  ThirdArg
#>         1         2         3
f(bce=10)
#>  FirstArg SecondArg  ThirdArg
#>         1         2        10
f(30,40)
#>  FirstArg SecondArg  ThirdArg
#>        30        40         3
f(bce=20,abc=10,100)
#>  FirstArg SecondArg  ThirdArg
#>        10       100        20
```

Advice on passing arguments to functions (Wickham 2019)

- Focus on positional mapping for the first one or two arguments
- Avoid positional mapping for arguments that are not used too often
- Unnamed arguments should come before named arguments.

The ... argument

- A final argument worth exploring is the ... argument, which will match any arguments not otherwise matched, and this can be easily forwarded to other functions
- It can also be converted to a list, to examine the arguments passed.
- Here is an example of how it can be used to pass any number of arguments to a function, and how the function can access these arguments.

```
test_dot1 <- function(...){
  ar = list(...)
  str(ar)
}
test_dot1(a=10,b=20:21)
#> List of 2
#>  $ a: num 10
#>  $ b: int [1:2] 20 21
```

Common use of the ... argument

However, the ... argument is often used to forward a set of arguments to another function, for example, here we can see how we can add flexibility to the function `test_dot2` by adding the argument ... as a parameter.

```
test_dot2 <- function(v,...){
  sum(v,...)
}
v <- c(1:3,NA)
test_dot2(v)
#> [1] NA
test_dot2(v,na.rm=TRUE)
#> [1] 6
```

Error checking for functions

- While functions are invaluable as small units of useful code, they must also be robust, and where an error is encountered, it should be highlighted.
- From a programming perspective, a decision needs to be made as to whether an error condition requires that the program be halted, or whether an error generates information that can be used as the program considers.
- We will take the first approach in this short example, and assume that the program must stop when an error in encountered, and this can be done uisng R's `stop()` function, which stops execution of the current expression, and executes an error action.
- The general process for creating robust functions is to test conditions early in the function, and so "fail fast" (Wickham 2019)

Error checks for `evens()` function

- Here, we return to the earlier example of filtering out the even values from a vector.
- For this case, we need to think about how we intend the function to be called, for example, what should happen if:
  - The vector is empty?
  - The vector is not an atomic vector?
  - The atomic vector is not numeric?

These would seeem like sensible checks to make before we would proceed with the function's core processing. In order to test whether or not a vector is empty, we can use the `length()` function to check this. For example:

```
v <- c() # an empty vector
length(v) == 0
#> [1] TRUE
```

Error checks for `evens()` function

Next, we need to make sure the vector is a numeric vector, for example, if a user sent in a character vector, it would not be possible to use the %% operator on a character vector.

```
v <- c("Hello", "World")
is.numeric(v)
#> [1] FALSE
```

Therefore, these two functions can be used to check the input values early, and "fail fast" if necessary. We add this logic to the early part of the function.

Adding error checks to evens()

```
evens <- function(v){
  if(length(v)==0)
    stop("Error>> exiting evens(), input vector is empty")
  else if(!is.numeric(v))
    stop("Error>> exiting evens(), input vector not numeric")
  v[v%%2==0]
}
```

Error checks

```r
# Robustness test 1, check for empty vector
t1 <- c()
evens(t1)
# Error in evens(t1) : Error>> exiting evens(), input vector
```

```r
# Robustness test 2, check for non-numeric vector
t2 <- c("This should fail")
evens(t2)
# Error in evens(t2) : Error>> exiting evens(), input vector
```

Error checks

```r
# Robustness test 3, check for non-atomic vector
t3 <- list(1:10)
evens(t3)
# Error in evens(t3) : Error>> exiting evens(), input vector r
```

```r
# Robustness test 4, should work ok
t4 <- 1:7
evens(t4)
#> [1] 2 4 6
```

Environments and Functions

- Understanding how environments work in key to figuring out how variables are accessed and retrieved in R.
- It is worth spending time an understanding environments, which is made up of two parts:
  - a frame (think of it as something like a list) that contain name-object bindings, and
  - a reference to a parent environment, which creates a *hierarchy of environments* within R.
- The global environment *R_GlobalEnv* is the interactive workspace that contains user-defined variables and functions.
- In showing the examples, we will make use of the library `pryr`, and in particular, a function called `where()`, which, for any given R object - expressed as a string -, will display the environment where it is located.

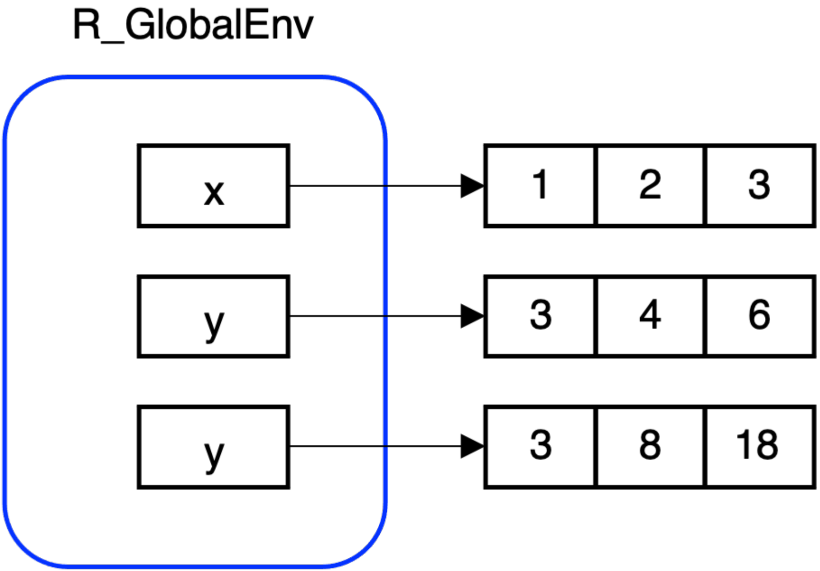
Consider the following example, where we assume that the workspace contains no variables. We define three variables, along with assignment statements.

Exploring environments

Consider the following example, where we assume that the workspace contains no variables. We define three variables, along with assignment statements.

```
suppressPackageStartupMessages(library(pryr))
x <- c(1,2,3)
y <- c(3,4,6)
z <- x * y
where("x")
#> <environment: R_GlobalEnv>
where("y")
#> <environment: R_GlobalEnv>
where("z")
#> <environment: R_GlobalEnv>
where("mtcars")
#> <environment: package:datasets>
#> attr(,"name")
```

Visualsing variables in `R_GlobalEnv`



Environments and Functions

- Environments are also highly important when understanding how functions work.
- When a function is created it obtains a reference (i.e. it "points to") the environment in which is was created, and this is known as the function's *enclosing environment*.
- The function `environment()` can be used to confirm a function's enclosing environment.

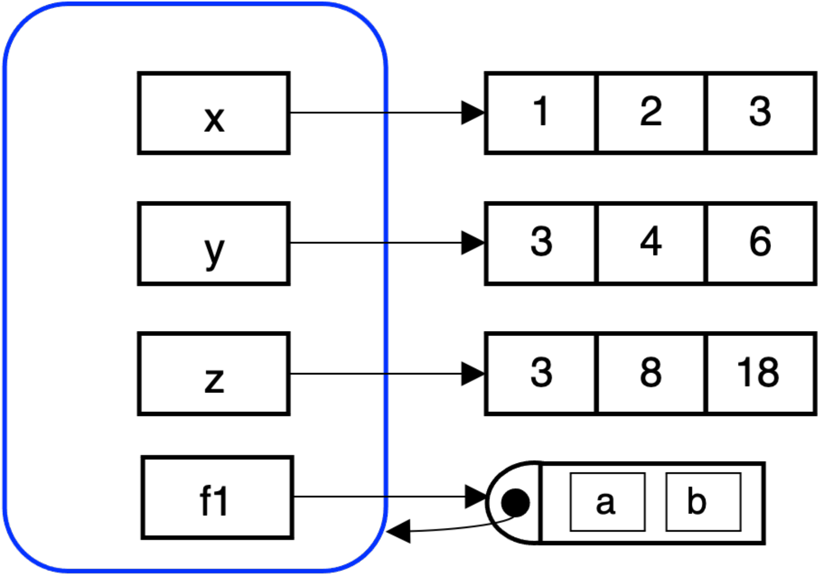
```
f1 <- function(a,b){
  (a+b)*c
}
environment(f1)
#> <environment: R_GlobalEnv>
```

Locating Variables from within functions. . .

How does the function find the value for `z`?

```
x <- c(1,2,3)
y <- c(3,4,6)
z <- x * y
f1 <- function(a,b){
  (a+b)*z
}
f1(10, 20)
#> [1]  90 240 540
```

Visualising Functions



Functions refer to their enclosing enviornment

- The diagram also shows an interesting feature of R, in that the function also contains a reference to its environment.
- This means that when the function *executes*, it also has a pathway to search its enclosing environment for variables and functions.
- For example, we can see that the function has the equation `(a+b)*z`, where `a` and `b` are local variables, and so are already part of the function.
- However, `z` is not part of the function, and therefore R will then search the enclosing environment to search for `z`, and if it finds it, will use that value in the calculation.
- If `z` cannot be found, an error results.

An overview of environments

- Environments form a tree structure, in which every environment has a parent environment, apart from the environment *at the top of the tree* which is known as the empty environment.
- Many of R's base functions are stored in the base environment (accessed using the function `baseenv()`), for example, functions such as `min()` and `max()`.
- Their location can be confirmed using the `where()` function.

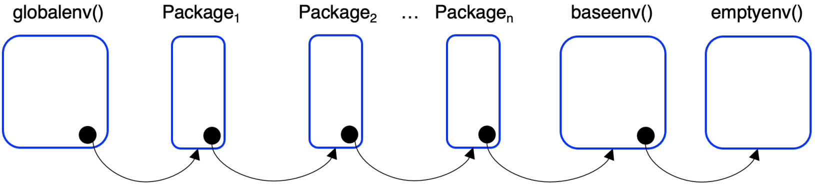


Figure 3: Environment structure in R

Exploring packages

```
where("min")
#> <environment: base>
where("max")
#> <environment: base>
```

- What is interesting is that a separate environment is also added for each new package loaded using `library()`
- The newest package's environment is added as the direct parent of the global environment.
- The full hierarchy can be easily shown using the function `search()`.
- Notice that the first environment shown is always the global environment, and the last environment shown is the base environment, as `search()` does not show the empty environment

Exploring packages via search()

```r
# Show the full hierarchy
search()
#>  [1] ".GlobalEnv"         "package:pryr"      "package:stats
#>  [4] "package:graphics"  "package:grDevices" "package:utils
#>  [7] "package:datasets"  "package:methods"   "Autoloads"
#> [10] "package:base"
# Show the empty environment
parent.env(baseenv())
#> <environment: R_EmptyEnv>
```

Useful example, note `base::max()` call

```
library(pryr)
(x <- 1:3)
#> [1] 1 2 3
max <- function(v){
  "Hello World"
}

where("max")
#> <environment: R_GlobalEnv>
(max(x))
#> [1] "Hello World"
(base::max(x))
#> [1] 3
```

Functionals with `lapply()`

- We have already demonstrated how the for loop can be used to iterate over a list, element by element.
- We now introduce a very important aspect of programming with R, which is the use of functionals, that take functions as part of their input, and use that function to process data.
- these functions can be used instead of loops to iterate over data and return a result.
- One of the most important functions that can be used to replace a loop is `lapply(x, f)`, which:
  - Accepts as input a list `x` and a function `f`
  - Returns as output a new list of the same length as `x`, where each element in the new list is the result of applying the function `f` to the corresponding element of the input list `x`.

The `lapply()` process - `my_lapply()`

```r
my_lapply <- function(x,f){
  # Create the output list vector
  o <- vector(mode="list",length = length(x))
  for(i in seq_along(x)){
    o[[i]] <- f(x[[i]])
  }
  o
}

l_in <- list(1:4,11:14,21:24)
l_out <- my_lapply(l_in,mean)
str(l_out)
#> List of 3
#>  $ : num 2.5
#>  $ : num 12.5
#>  $ : num 22.5
```

Summary of the process

- Two inputs are passed to the function, the data (a list of 3 elements), and the function to apply to the elements, which in this example is the function `mean()`.
- Inside the function, the list `l_in` is mapped to the variable `x`, and the function `mean` is mapped to `f`.
- The first action is to create a variable (`o`) that will eventually store the result. We know that the number of elements in this output variable must be the same as the number of elements on the input list, as that's the key idea behind the apply functional, it operates on each element with the same function, and returns all the results.
- The function then iterates over the entire input list (now stored in `x`), and calls the function `f` with this data, storing the result of this calculation in the corresponding location of the variable `o`.
- Once all the list elements have been processed, the variable `o` is returned, which is a list of three elements, and each element contains the mean of the corresponding input list element.

Using Base R's `lapply()`

- So while it's useful to see how the lapply process works, there is no need to duplicate the function by writing your own version.
- The code below shows the solution using `lapply()`.

```
l_in <- list(1:4,11:14,21:24)
l_out <- lapply(l_in,mean)
str(l_out)
#> List of 3
#>  $ : num 2.5
#>  $ : num 12.5
#>  $ : num 22.5
```

In later chapters, when we introduce the `purrr` package, which provides a set of functions that can be used to iterate over data structures, in a similar way to `lapply()`, but with more functionality.

Example using repurrrsive

```
library(repurrrsive)
# Search for movies by George Lucas and store these in a new l
target      <- "George Lucas"
# Call lapply to return a list of logical vectors
is_target   <- lapply(sw_films,function(x)x$director==target)
# Convert this list to an atomic vector, which is needed for f
is_target   <- unlist(is_target)
# Filter the list to contain the George Lucas movies (4)
target_list <- sw_films[is_target]
```

Additional Example

```
# Search for movies by George Lucas and store these in a new l
target      <- "George Lucas"
target_list <- lapply(sw_films,function(x)if(x$director==targe
target_list <- target_list[!is.na(target_list)]
```

With the filtered list, you can then call `lapply()` to return the movie titles.

```
# Get the movie titles as a list
movies <- lapply(target_list,function(x)x$title)
movies <- unlist(movies)
movies
#> [1] "A New Hope"              "Attack of the Clones" "The Phan
#> [4] "Revenge of the Sith"
```

Creating a different list structure

```
# Create a new list to store the data in a different way
sw_films1               <- list(title=c(),
                                episode_id=c(),
                                director=c())
sw_films1$title         <- unlist(lapply(sw_films,
                                         function(x)x$title))
sw_films1$episode_id <- unlist(lapply(sw_films,
                                         function(x)x$episode_id
sw_films1$director   <- unlist(lapply(sw_films,
                                         function(x)x$director))
str(sw_films1)
#> List of 3
#>  $ title     : chr [1:7] "A New Hope" "Attack of the Clone
#>  $ episode_id: int [1:7] 4 2 1 3 6 5 7
#>  $ director  : chr [1:7] "George Lucas" "George Lucas" "Ge
```

Planet Diameters

```
# Return all planet diameters as numeric, with NA where it's u
diameters <- unlist(lapply(sw_planets,
                           function(x)
                             if (x$diameter != "unknown")
                                 as.numeric(x$diameter)
                             else NA))
# Provide a summary of the data
summary(diameters)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#>       0    7812   11015   12388   13422  118000      17
```

The native pipe operator in R, (|>)

- |> allows you to chain a number of operations together, without having to assign intermediate variables.
- This operator, originally based on the %>% operator from the package `magrittr`, allows you to construct a data processing pipeline.
- The key idea is that the output from one step becomes the input to the second step, and it provides an elegant way to assemble the different steps.
- The general format of the pipe operator is `LHS |> RHS`, where `LHS` is the first argument of the function defined on the `RHS`.

```
1:3 |> sqrt()
#> [1] 1.000000 1.414214 1.732051
```

More detailed example...

- Let's say we want to get the minimum of a randomised vector value (we will use the function `runif()` for this, which generates a random uniformly distributed number in the range 0-1.)
- With the |> operator, we first identify the LHS data for `min()`, and then we insert the function `min()` with no arguments as the RHS of the expression

```
set.seed(200)
# Generate a vector  of random numbers
n1 <- runif(n = 10)
# Show the minimum the usual way
min(n1)
#> [1] 0.09650122
# Use the native pipe to isolate the input, and the "pipe" it
n1 |> min()
#> [1] 0.09650122
```

Adding Additional Transformations

- More operations can be added to the chain, and it that case, the output from the first RHS then becomes the LHS for the next operation.
- For example, we could also add an addition operation to the example, to round the number of decimal places to 3, by using the round() function.

```
n1 |> min() |> round(3)
#> [1] 0.097
```

Example using `mtcars` data frame

We will use the data frame `mtcars` to perform the following chain of data transformations

- Take as input `mtcars`
- Convert this to a list, using the function `as.list`
- Process the list one element at a time, and get the average value of each variable
- Convert the list returned by `lapply()` to an atomic vector (using `unlist()`)
- Store the result in a variable

Solution using |>

```r
a1 <- mtcars                          |>
      as.list()                       |>
      lapply(function(x)mean(x))  |>
      unlist()
a1
#>        mpg        cyl       disp         hp       drat         3.
#>  20.090625   6.187500 230.721875 146.687500   3.596563   3.
#>         vs         am       gear       carb
#>   0.437500   0.406250   3.687500   2.812500
```

Useful R Functions (1/2)

R Function	Description
`duplicated()`	Identifies the elements of a vector that are duplicates and returns a logical vector
`pryr::where()`	Returns the environment in which a name (as a string) is defined
`environment()`	Can be used to find the environment for a function
`parent.env()`	Finds the parent environment for a given input environment
`search()`	Returns a vector of environment names starting a the R_GlobalEnv
`globalenv()`	Returns a reference to the global environment
`baseenv()`	Returns a reference to the base environment

Useful R Functions (2/2)

R Function	Description
`lapply(x,f)`	A functional that applies a function `f` to each element of `x` and returns the results in a list
`stop()`	

Exercise 1 - Return even numbers

Write a function `get_even1()` that returns only the even numbers from a vector. Make use of R's modulus function %% as part of the calculation. Try and implement the solution as one line of code. The function should transform the input vector in the following way.

```
set.seed(200)
(v <- sample(1:20,10))
#>  [1]  6 18 15  8  7 12 19  5 10  2
(v1 <- get_even1(v))
#> [1]  6 18  8 12 10  2
```

Exercise 2 - Adding extra argument

Write a similar function `get_even2()` that takes a second parameter `na.omit`, with a default of `FALSE`. If `na.omit` is set to `TRUE`, the vector is pre-processed in the function to remove all `NA` values before doing the final calculation.

```
set.seed(200)
v <- sample(1:20,10)
i <- c(1,5,7)
v[i] <- NA
v
#>  [1] NA 18 15  8 NA 12 NA  5 10  2
(v1 <- get_even2(v))
#> [1] NA 18  8 NA 12 NA 10  2
(v2 <- get_even2(v,na.omit=TRUE))
#> [1] 18  8 12 10  2
```

Exercise 3: Using `lapply()`

Use `lapply()` followed by an appropiate post-processing function call, to generate the following output (median's of vectors), based on the input list.

```
# Create the list that will be processed by lapply
l1 <- list(a=1:5,b=100:200,c=1000:5000)
```

```
# The result is stored in ans
ans
#>    a    b    c
#>    3  150 3000
```

Lecture Summary

- How to write your own function, call it via arguments, and return values based on the last evaluated expression.
- The different ways to pass arguments to functions.
- How to add robustness checking to your functions to minimise the chance of processing errors.
- What an environment is, and the environment hierarchy within R.
- How a function contains a reference to its enclosing environment and how this is used to access variables in the global environment.
- What a functional is, namely, a function that can take another function as an argument.
- The functional `lapply()`, and how this can be used, as an alternative to loops, to iterate over lists and vectors.
- R's native pipe operator, and how this can be used to streamline a sequence of data processing operations.
- How to solve all three test exercises.