CT5102: Programming for Data Analytics

3. Functions, Functionals, and the R Pipe

Prof. Jim Duggan,
School of Computer Science
University of Galway.

https://github.com/JimDuggan/explore_or

An important advantage of R and other interactive languages is to make programming in the small an easy task. You should exploit this by creating functions habitually, as a natural reaction to any idea that seems likely to come up more than once.

— John Chambers (Chambers, 2017)

(3.1) Functions

- A function can be defined as a group of instructions that: takes input uses the input to compute other values, and returns a result (Matloff, 2011)
- 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
  - arguments provides the arguments (inputs) to a function, and are separated by commas
  - expression is any legal R expression contained within the function body, and is usually enclosed in curly braces (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 – returning even numbers

- First, we explore the logic in the console
- R's modulus operator %% is used, as this returns the remainder of two numbers, following their division.
- This logic will then be embedded within an R function which we will call evens(), which takes in one argument (the original vector), and returns a filtered version of the vector that only includes even numbers

```
v <- 1:5
x <- v %% 2
x
#> [1] 1 0 1 0 1
# The logical vector where even values are TRUE
lv <- x == 0

# Show the logical vector
lv
#> [1] FALSE  TRUE FALSE  TRUE FALSE

# Filter the original vector
v[lv]
#> [1] 2 4
```

The (short) function

```
evens <- function(v){
  v[v%%2==0]
}

x1 <- 1:7

evens(x1)
#> [1] 2 4 6
```

A second function – removing duplicates

- We use an existing base R function to create a new one
- Our 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
- We can invert this output to achieve our desired result.

```
set.seed(100)
v <- sample(1:6,10,replace = T)
v
#>  [1] 2 6 3 1 2 6 4 6 6 4
duplicated(v)
#>  [1] FALSE FALSE FALSE FALSE  TRUE
v[!duplicated(v)]
#> [1] 2 6 3 1 4
```

```r
my_unique <- function(x){
  # Use duplicated() to create a logical vector
  dup_logi <- duplicated(x)
  # Invert the logical vector so that unique values are set to TRUE
  unique_logi <- !dup_logi
```

```r
  # Subset x to store those values are unique
  ans <- x[unique_logi]
  # Evaluate the variable ans so that it is returned
  ans
}
```

```r
# The call to source loads the function into the global environment
source("my_functions.R")
```

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

Functionals

- In R, functions are objects, so they can be passed to functions as arguments
- Functionals are functions that accept functions as arguments.
- To send a function as an argument, all that is required is the function name.

```r
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
```

Functions can be anonymous (no variable binding)

```r
# Call my_summary() using an anonymous function
my_summary(1:10,function(y)min(y))
#> [1] 1
# Call my_summary() using an anonymous function
my_summary(1:10,function(y)max(y))
#> [1] 10
```

(3.2) 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).
- 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 have default values.

sum {base}

Sum of Vector Elements

Description

`sum` returns the sum of all the values present in its arguments.

Usage

```
sum(..., na.rm = FALSE)
```

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

Passing arguments…

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

- By position, where the arguments are copied to the corresponding argument location
- By complete name, where arguments are first copied to their corresponding name, before other arguments are then copied via their positions.
- By partial name, where argument names are matched, and where a unique match is found, that argument will be selected

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

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

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

Default values

- arguments 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 an arbitrary default value.

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

```r
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
```

General Guidelines for passing arguments

- Hadley Wickham (Wickham, 2019) provides valuable advice for passing arguments to functions, for example:
    1. to focus on positional mapping for the first one or two arguments
    2. to avoid positional mapping for arguments that are not used too often, and,
    3. unnamed arguments should come before named arguments.

(3.3) Error Checking within functions

- While functions are invaluable as small units of useful code, they must also be robust.
- When an error is encountered, it should be dealt with.
- 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 relayed to the user
- We can use R's stop() function, which stops execution of the current expression, and executes an error action

Our earlier example (first function)

- what should happen if the input vector:
  - Is empty?
  - Is not numeric?
- We need to add logic for these scenarios and “exit gracefully”
- Two checks can help

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

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

```r
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]
}
```

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

```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 not numeric
```

(3.4) Environments and Functions

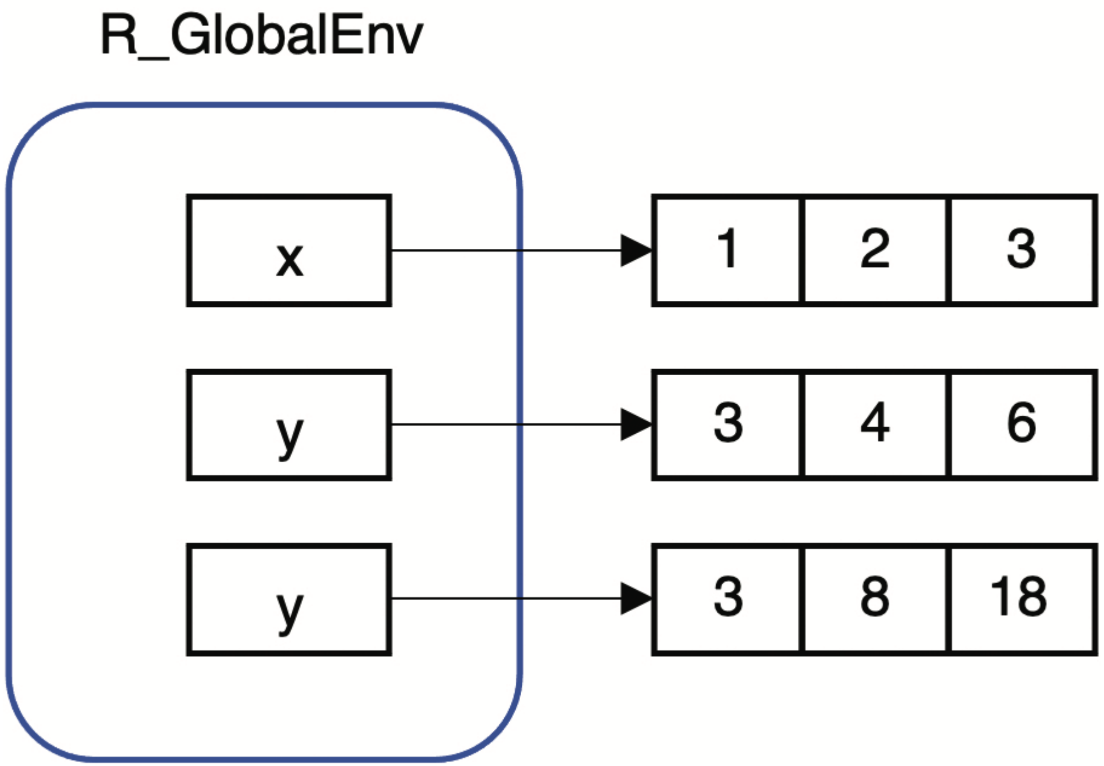
- Understanding how environments work is key to figuring out how variables are accessed in R.
- It is worth spending time on understanding an environment, which comprises two parts:
  1. a frame that contains name-object bindings, and
  2. a reference to its parent environment.
- This reference mechanism creates a hierarchy of environments within R.
- The global environment R_GlobalEnv, is the interactive workspace that contains user-defined variables and functions

pryr::where()

```r
library(pryr)
x <- c(1,2,3)
y <- c(3,4,6)
z <- x * y
pryr::where("x")
#> <environment: R_GlobalEnv>
pryr::where("y")
#> <environment: R_GlobalEnv>
pryr::where("z")
#> <environment: R_GlobalEnv>
```

Visualisation

- The values of a variable are stored in memory, and for a vector, these are in successive (i.e., contiguous) locations. We can visualize this storage as an array structure of three cells for each of the variables x, y, and z.
- We also need a variable name (identifier) that references (or “points to”) the values in memory, and the link between the variable and its value in memory is known as a binding.



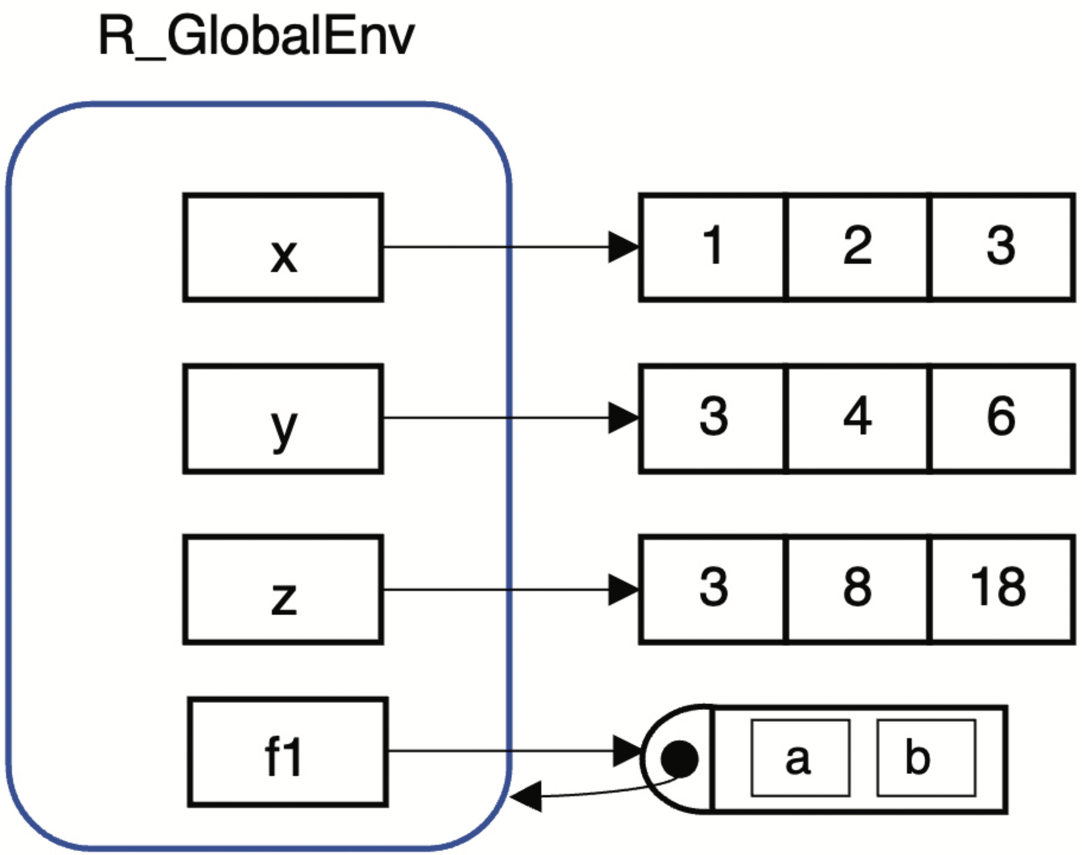
Environment and Functions

- Environments are also important for understanding how functions work.
- When a function is created, it obtains a reference (i.e., it “points to”) the environment in which it 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)*z
}
environment(f1)
#> <environment: R_GlobalEnv>
```

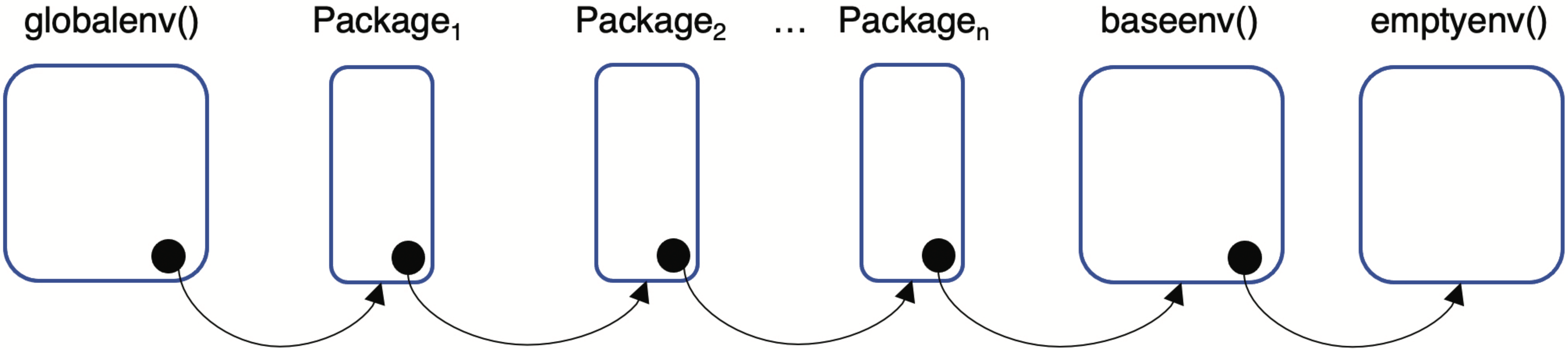
A function links to its enclosing environment

- The function also contains a reference to its enclosing environment
- This means that when the function executes, it also has a pathway to search its enclosing environment for variables and functions.
- If a variable is not in the function, the enclosing environment is then searched.



The environment structure in R: *a tree structure*

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



The *superassignment* operator

- While the function f1() can read the variable z, it cannot change the value of z using the <- operator.
- However, in R there is another operator known as the superassignment operator <<-, and this can be used within functions to modify a variable in the parent environment.
- If the variable does not exist in the parent environment, then a variable will be created in the global environment.

```r
c <- 20
f2 <- function(a,b){
  ans <- a + b + c
  c <<- 100
  ans
}
```

```r
c
#> [1] 20
d <- f2(2,4)
d
#> [1] 26
c
#> [1] 100
```

(3.5) Functionals with lapply()

- In the previous lecture, we demonstrated how the for loop can be used to iterate over a list, element by element.
- We now introduce an important aspect of programming with R, which is the use of functionals, which take functions as part of their input, and use that function to process data.
- In many cases, functionals can be used instead of loops to iterate over data and return a result.

lapply() – overall logic

- One of the most important functionals that can be used is lapply(x, f), which:
  - Accepts as input a list x and a function f,
  - Returns as output a new list of exactly 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.

```
my_lapply <- function(x,f){
  # Create the output list vector
  o <- vector(mode="list",length = length(x))
  # Loop through the entire input list
  for(i in seq_along(x)){
    # Apply the function to each element and
    # store in the corresponding output location
    o[[i]] <- f(x[[i]])
  }
  # Return the output list
  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
```

Using lapply()

```r
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
```

(3.6) Mini-case "Star Wars" with functionals

- To remind ourselves, the goal is to find the movies directed by George Lucas.
- We will use lapply() to identify which elements are relevant.
- lapply() takes two arguments:
  – The list sw_films, containing seven elements, where each element is itself a list of 14 elements, one of which is the movie director.
  – An anonymous function that takes in each successive list element (parameter x), and compares the $director element with the variable target, which is defined in the global environment, and so can be accessed by the anonymous function.
  – The anonymous function returns the last evaluated expression, which is the result of the relational expression (either TRUE or FALSE).

```r
library(repurrrsive)
# Search for movies by George Lucas and store these in a new list
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 filtering
is_target   <- unlist(is_target)

# Filter the list to contain the George Lucas movies
target_list <- sw_films[is_target]
length(target_list)
#> [1] 4
```

Another way…

```r
# Search for movies by George Lucas and store these in a new list
target      <- "George Lucas"
target_list <- lapply(sw_films,function(x)
                                 if(x$director==target) x
                                   else NA)
target_list <- target_list[!is.na(target_list)]
length(target_list)
#> [1] 4
# Get the movie titles as a list
movies <- lapply(target_list,function(x)x$title)
movies <- unlist(movies)
movies
#> [1] "A New Hope"          "Attack of the Clones"
#> [3] "The Phantom Menace"  "Revenge of the Sith"
```

R's Pipe Operator |>

- The native pipe operator in R, represented by the symbol |>, 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 (Bache and Wickham, 2014), allows you to construct a data processing pipeline, where an input is identified (e.g., a list, vector, or later in the book, a data frame), an output specified, and each discrete step in generating the output is linked together in a chain.
- The general format of the pipe operator is LHS |> RHS, where LHS is the first argument of the function defined on the RHS.

Example

```r
set.seed(200)
# Generate a vector  of random numbers
n1 <- runif(n = 10)
# Show the minimum the usual way
min(n1)
#> [1] 0.0965
# Use the native pipe to generate the same answer
n1 |> min()
#> [1] 0.0965
```

Second Example

- Take as input mtcars (Environment package::datasets)
- Convert mtcars to a list, using the function as.list(). Note that data frames are technically a list.
- Process the list one element at a time, and get the average value of each list element
- Convert the list returned by lapply() to an atomic vector (using unlist())
- Store the result in a variable.

R’s pipe in action…

```r
a1 <- mtcars                           |>  # The input
      as.list()                        |>  # Convert to a list
      lapply(function(x)mean(x))       |>  # Get the mean of each element
      unlist()                             # Convert to atomic vector
a1
#>       mpg      cyl     disp       hp     drat       wt     qsec
#>   20.0906   6.1875 230.7219 146.6875   3.5966   3.2172  17.8487
#>        vs       am     gear     carb
#>    0.4375   0.4062   3.6875   2.8125
```

(3.7) Summary Functions

Function	Description
duplicated()	Identifies duplicates in a vector.
where()	Returns the environment for a variable (pryr library).
environment()	Finds the environment for a function.
library()	Loads and attaches add-on packages.
parent.env()	Finds the parent environment for an input environment.
search()	Returns a vector of environment names.
globalenv()	Returns a reference to the global environment.
baseenv()	Returns a reference to the base environment.
lapply(x,f)	Applies f to each element of x and returns results in a list.
rm()	Removes an object from an environment.
stop()	Stops execution of the current expression.
unique()	Returns a vector with duplicated elements removed.

(3.8) Exercises

1. 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 to 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)
v
#>  [1]  6 18 15  8  7 12 19  5 10  2
v1 <- get_even1(v)
v1
#> [1]  6 18  8 12 10  2
```

2. 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)
v1
#> [1] NA 18  8 NA 12 NA 10  2
v2 <- get_even2(v,na.omit=TRUE)
v2
#> [1] 18  8 12 10  2
```

4. What will be the output from the following function call?

```
a <- 100

env_test <- function(b,c=20){
  a+b+c
}

env_test(1)
```

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

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

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