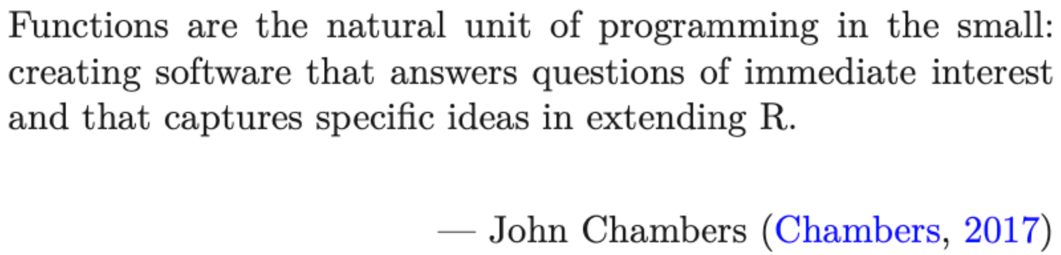


Functions are the natural unit of programming in the small: creating software that answers questions of immediate interest and that captures specific ideas in extending R.

— John Chambers (Chambers, 2017)

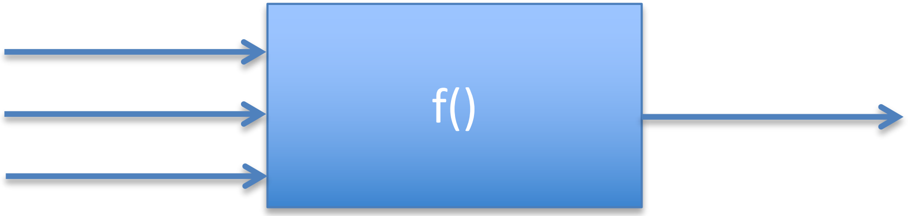
Data Science for Operational Researchers using R

02 – Functions, Lists and Functionals

https://github.com/JimDuggan/Data-Science-for-OR

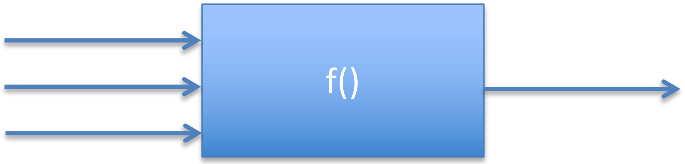
(1) Functions

- 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 building blocks in R
- We already have used many R inbuilt functions: sample(), table()
- Now we can write our own in a source file.



General Form

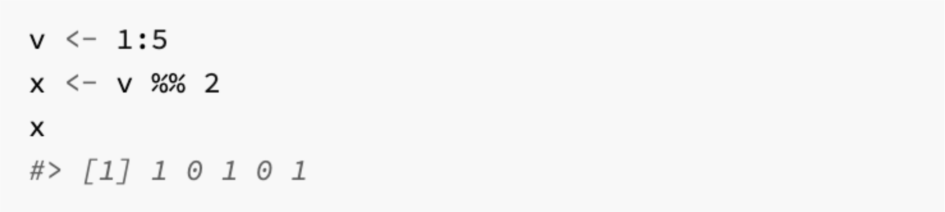
function(*arguments*)
    expression



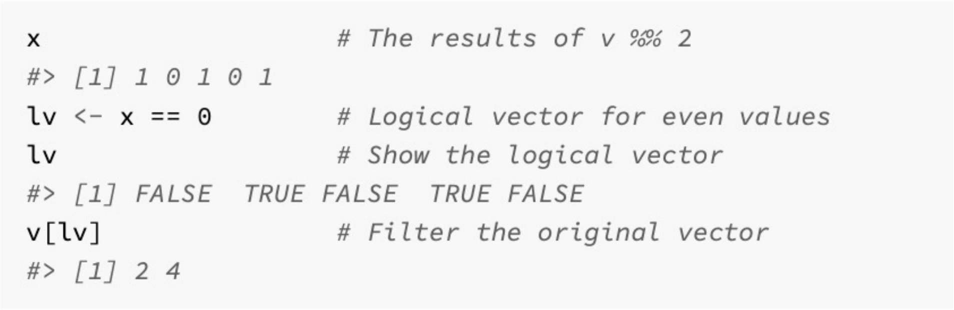
- *arguments* gives the arguments, separated by commas.
- *Expression* (body of the function) is any legal R expression, usually enclosed in { }
- **Last evaluated expression is returned**
- return() can also be used, but usually for exceptions.

A first function – returning even numbers

- 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
- We will focus on the data processing first, and then encapsulate this within a function



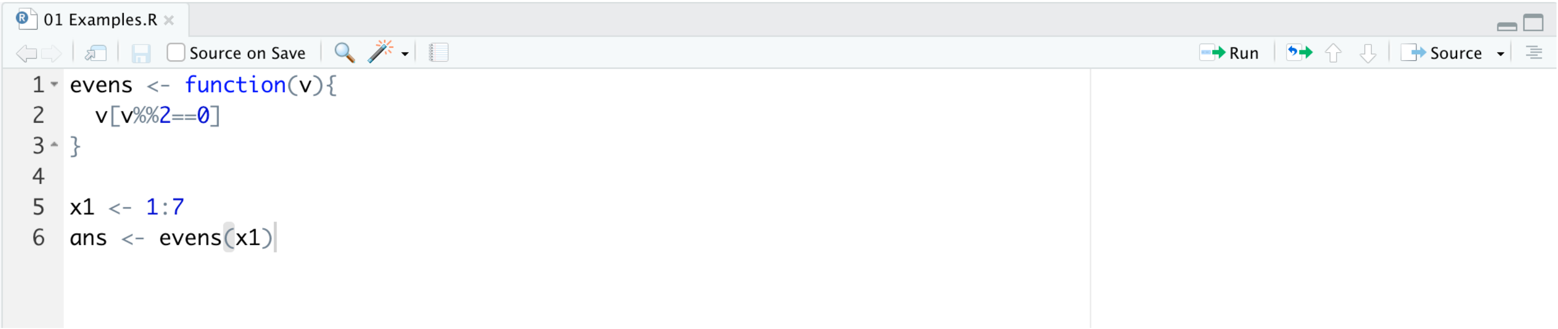
```
v <- 1:5
x <- v %% 2
x
#> [1] 1 0 1 0 1
```



```
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
```

The function, and a source file

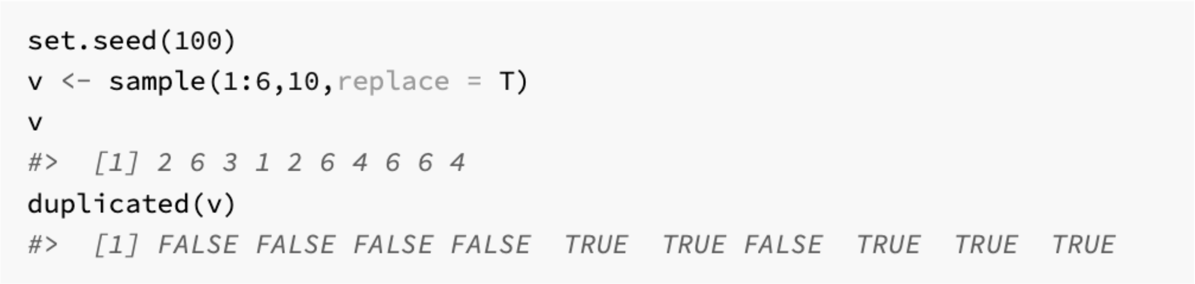
```
evens <- function(v){
  v[v%%2==0]
}
x1 <- 1:7
evens(x1)
#> [1] 2 4 6
```



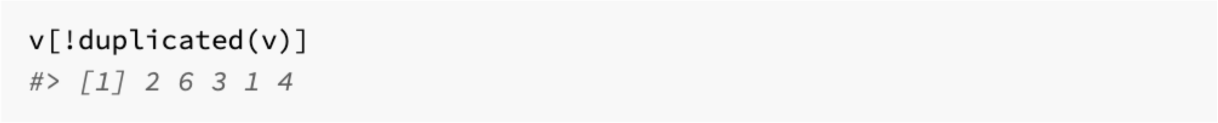
```
evens <- function(v){
  v[v%%2==0]
}

x1 <- 1:7
ans <- evens(x1)
```

A second function – removing duplicates

- This builds on the work of others
- duplicated() function
- Will use this to identify duplicates so they can be removed within a new function

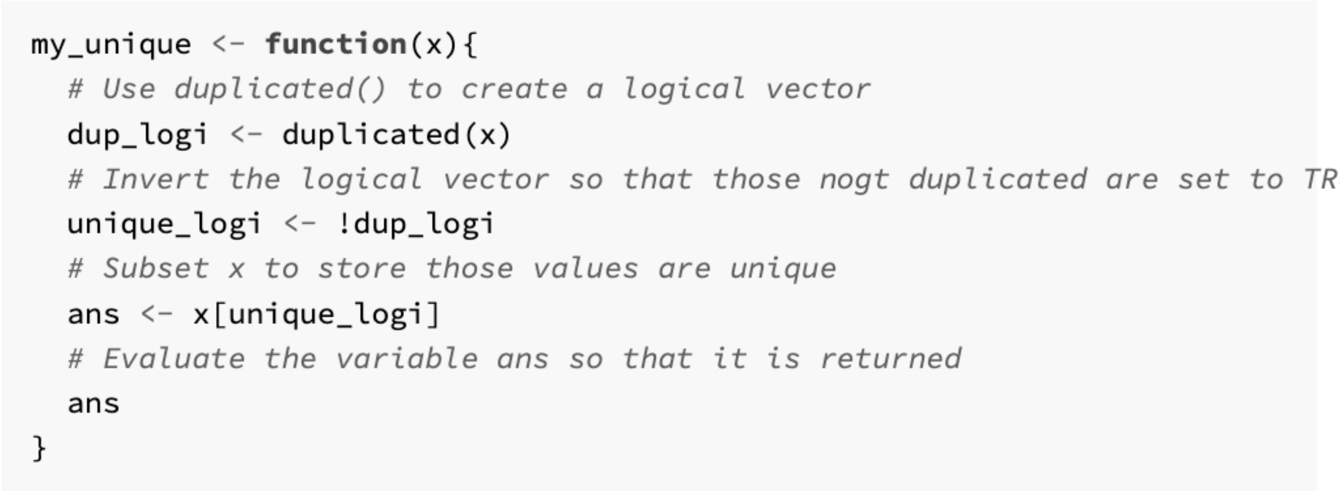


```
set.seed(100)
v <- sample(1:6,10,replace = T)
v
#>  [1] 2 6 3 1 2 6 4 6 6 4
duplicated(v)
#>  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

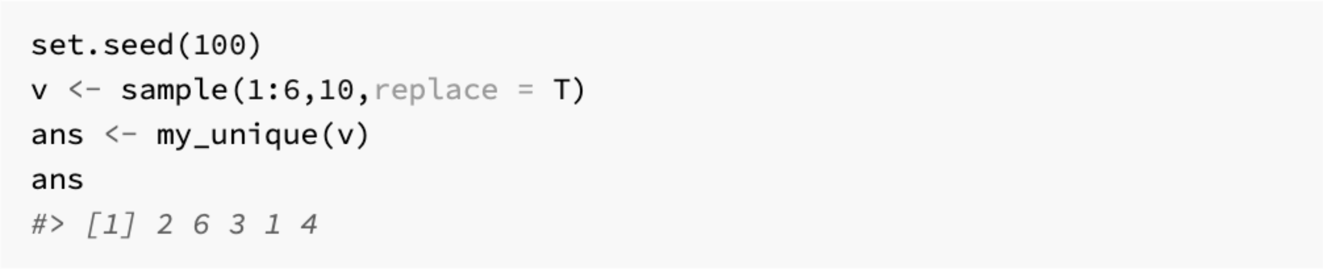


```
v[!duplicated(v)]
#> [1] 2 6 3 1 4
```

Function Code

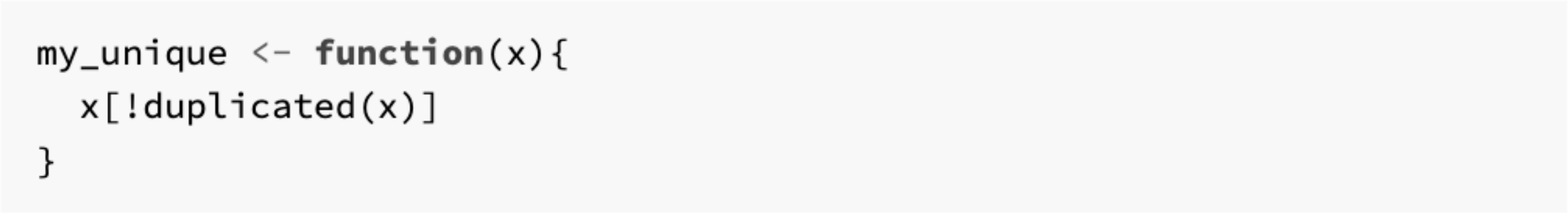


```r
my_unique <- function(x){
  # Use duplicated() to create a logical vector
  dup_logi <- duplicated(x)
  # Invert the logical vector so that those nogt duplicated are set to TRUE
  unique_logi <- !dup_logi
  # Subset x to store those values are unique
  ans <- x[unique_logi]
  # Evaluate the variable ans so that it is returned
  ans
}
```



```r
set.seed(100)
v <- sample(1:6,10,replace = T)
ans <- my_unique(v)
ans
#> [1] 2 6 3 1 4
```

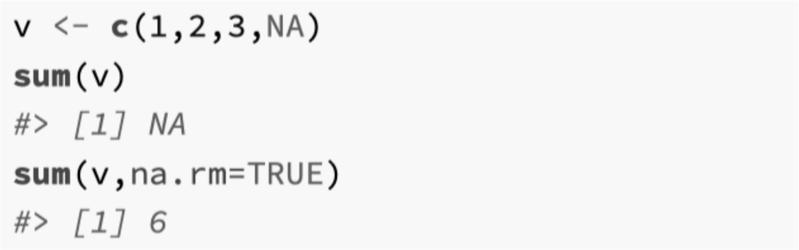
A reduced size version…



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

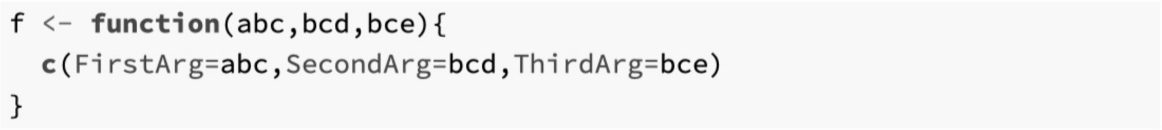
Passing arguments to functions

- 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.



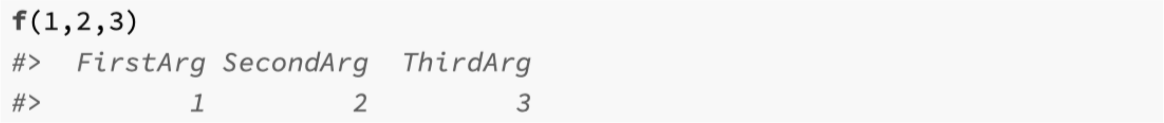
```
v <- c(1,2,3,NA)
sum(v)
#> [1] NA
sum(v,na.rm=TRUE)
#> [1] 6
```

Important ideas for passing arguments



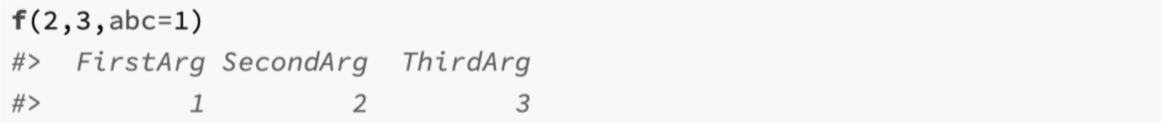
```
f <- function(abc,bcd,bce){
  c(FirstArg=abc,SecondArg=bcd,ThirdArg=bce)
}
```

- By position



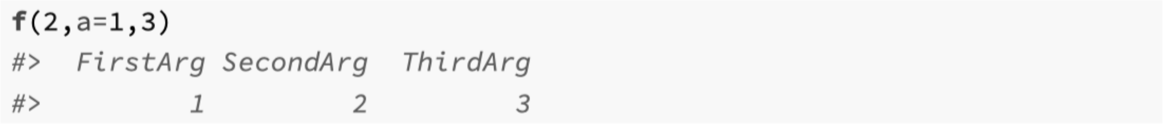
```
f(1,2,3)
#>  FirstArg SecondArg  ThirdArg
#>         1         2         3
```

- By complete name



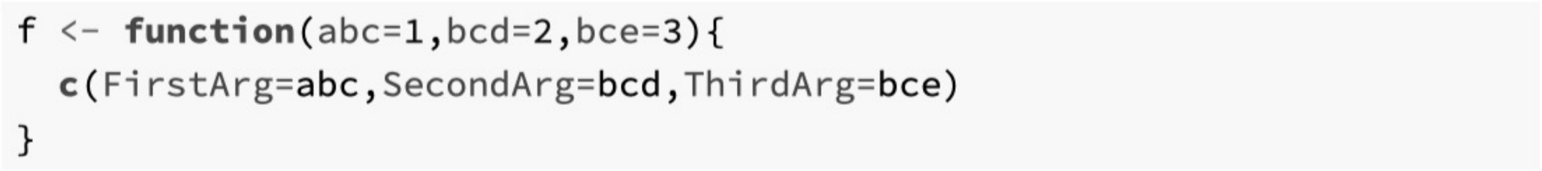
```
f(2,3,abc=1)
#>  FirstArg SecondArg  ThirdArg
#>         1         2         3
```

- By partial name

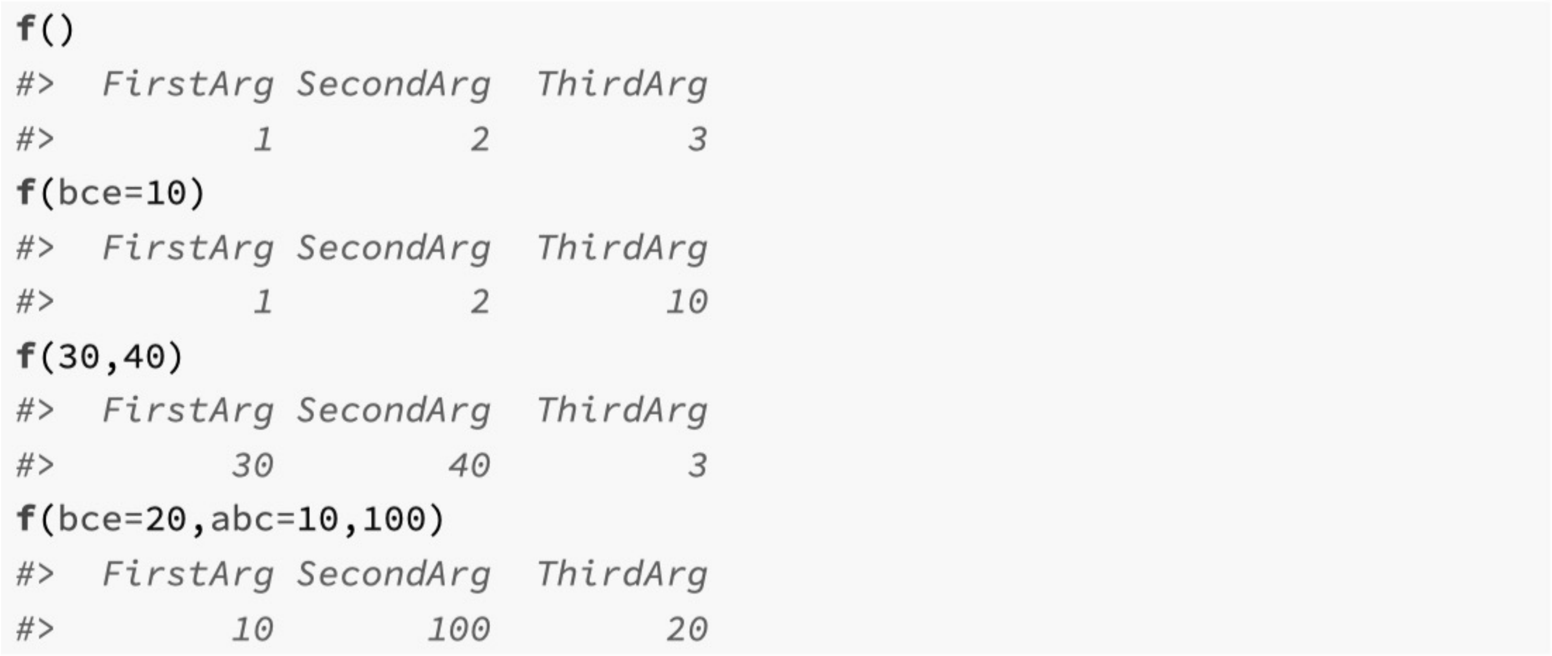


```
f(2,a=1,3)
#>  FirstArg SecondArg  ThirdArg
#>         1         2         3
```

Default values



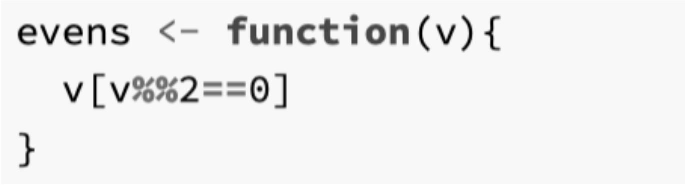
```
f <- function(abc=1,bcd=2,bce=3){
  c(FirstArg=abc,SecondArg=bcd,ThirdArg=bce)
}
```



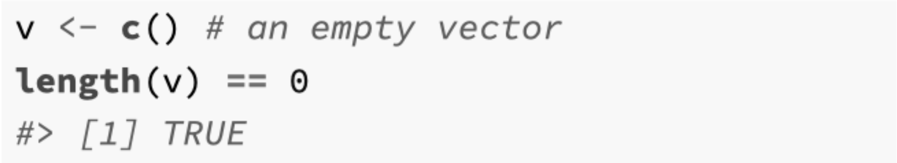
```
f()
#>   FirstArg SecondArg  ThirdArg
#>          1         2         3
f(bce=10)
#>   FirstArg SecondArg  ThirdArg
#>          1         2        10
f(30,40)
#>   FirstArg SecondArg  ThirdArg
#>         30        40         3
f(bce=20,abc=10,100)
#>   FirstArg SecondArg  ThirdArg
#>         10 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

Creating Robust Functions

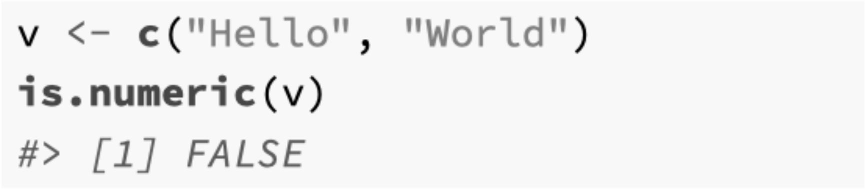
- The general process for creating robust functions is to test conditions early in the function, and so “fail fast” (Wickham, 2019).
- Filtering even values function
  - The vector is empty?
  - The vector is not an atomic vector?
  - The atomic vector is not numeric?



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

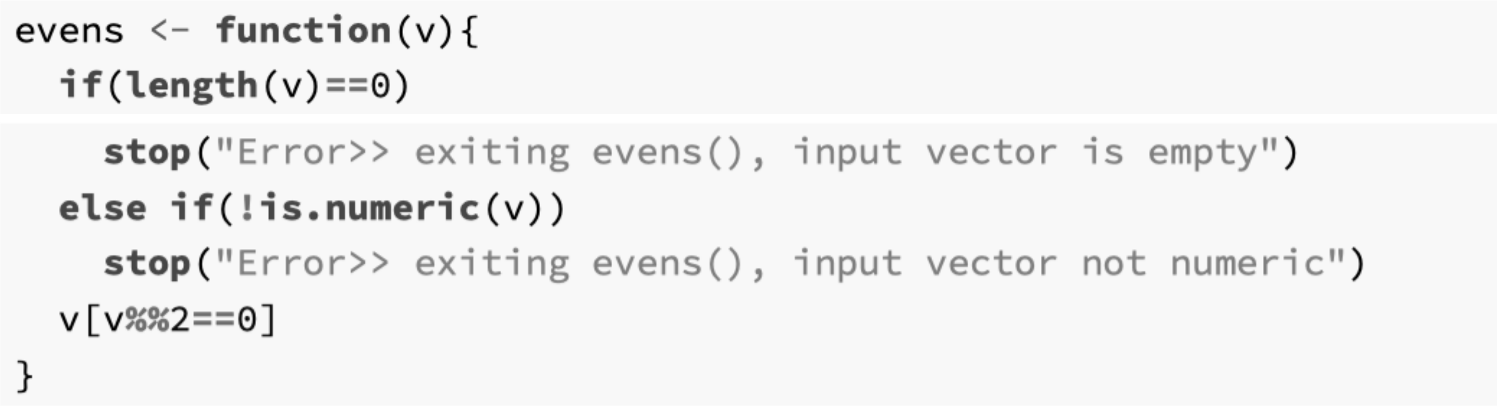


```
v <- c() # an empty vector
length(v) == 0
#> [1] TRUE
```

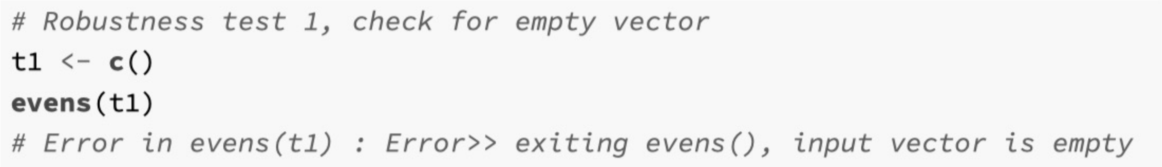


```
v <- c("Hello", "World")
is.numeric(v)
#> [1] FALSE
```

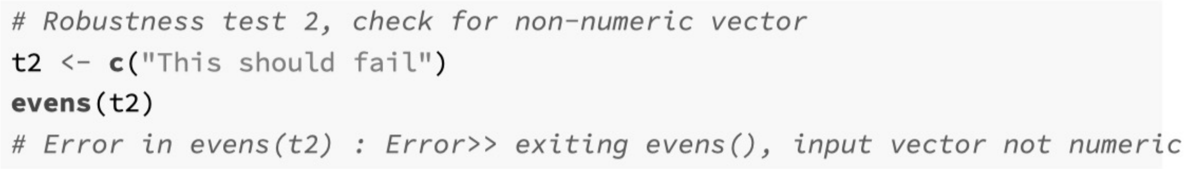
Solution



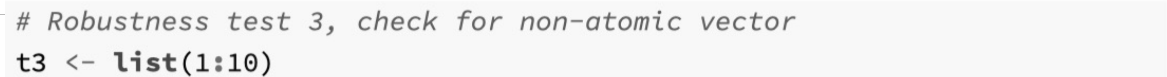
```
evens <- function(v){
  if(length(v)==0)
    stop("Error>> exiting evens(), input vector is empty")
  else if(!is.numeric(v))
    stop("Error>> exiting evens(), input vector not numeric")
  v[v%%2==0]
}
```



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

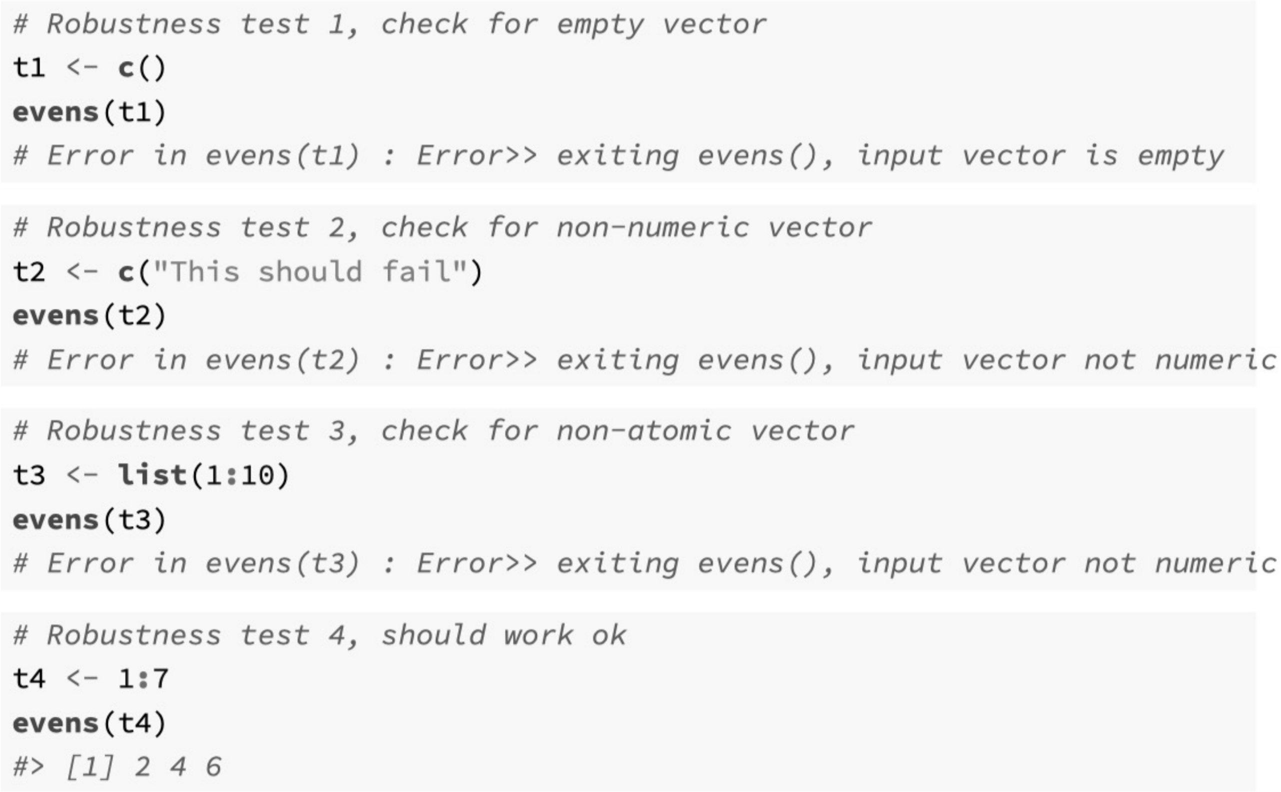


```
# Robustness test 2, check for non-numeric vector
t2 <- c("This should fail")
evens(t2)
# Error in evens(t2) : Error>> exiting evens(), input vector not numeric
```



```
# Robustness test 3, check for non-atomic vector
t3 <- list(1:10)
```

Solution



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

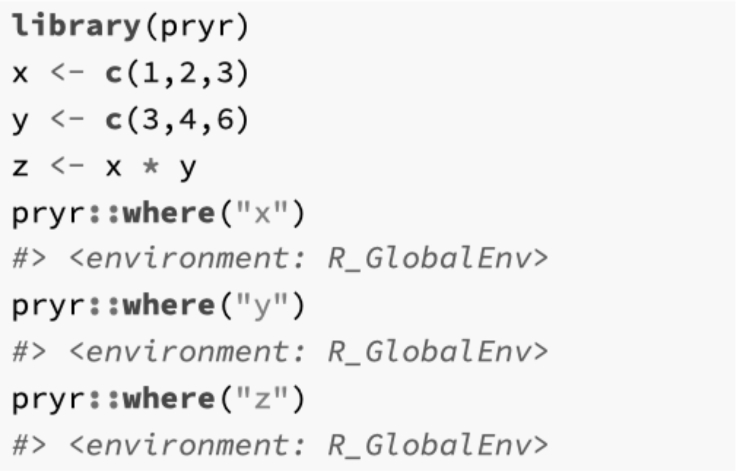
```
# Robustness test 2, check for non-numeric vector
t2 <- c("This should fail")
evens(t2)
# Error in evens(t2) : Error>> exiting evens(), input vector not numeric
```

```
# Robustness test 3, check for non-atomic vector
t3 <- list(1:10)
evens(t3)
# Error in evens(t3) : Error>> exiting evens(), input vector not numeric
```

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

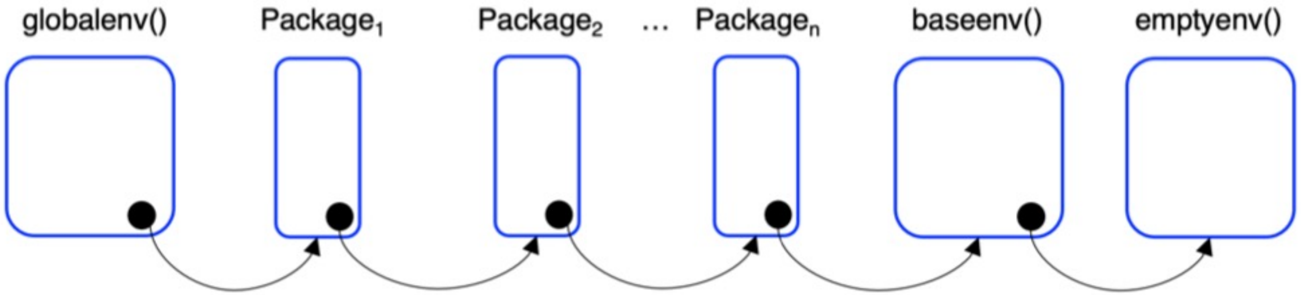
Environments

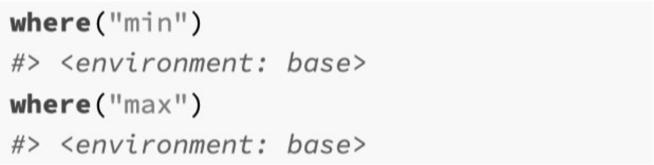
- Understanding how environments work in key to figuring out how variables are accessed and retrieved in R.
- Environments are made up of two parts: (1) a frame (think of it as something like a list) that contain name-object bindings, and (2) a reference to a parent environment, which creates a hierarchy of environments within R



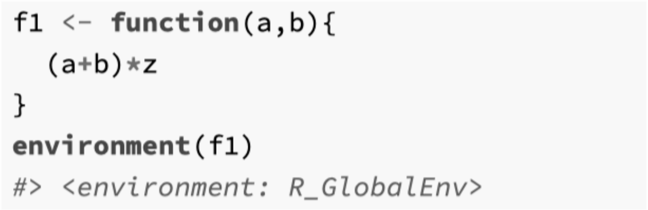
```
library(pryr)
x <- c(1,2,3)
y <- c(3,4,6)
z <- x * y
pryr::where("x")
#> <environment: R_GlobalEnv>
pryr::where("y")
#> <environment: R_GlobalEnv>
pryr::where("z")
#> <environment: R_GlobalEnv>
```

Environment Structure in R

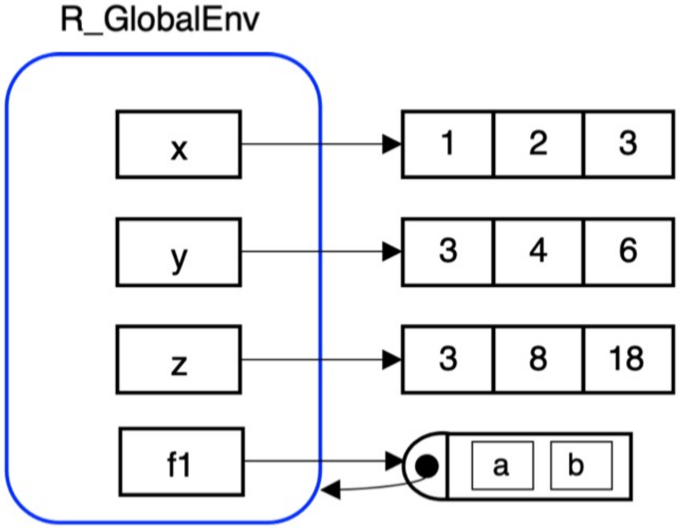




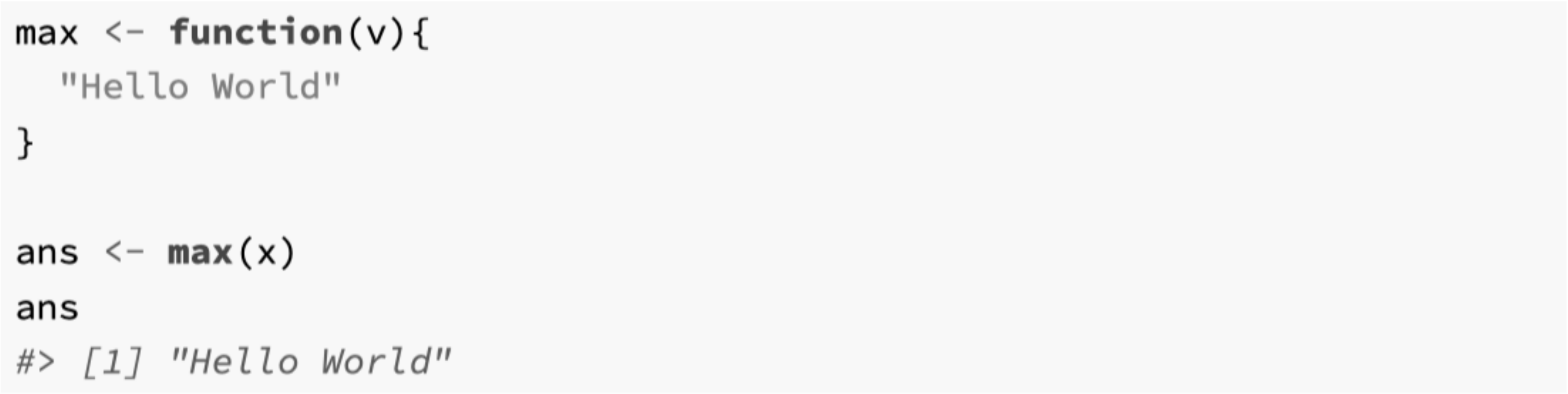
```
where("min")
#> <environment: base>
where("max")
#> <environment: base>
```



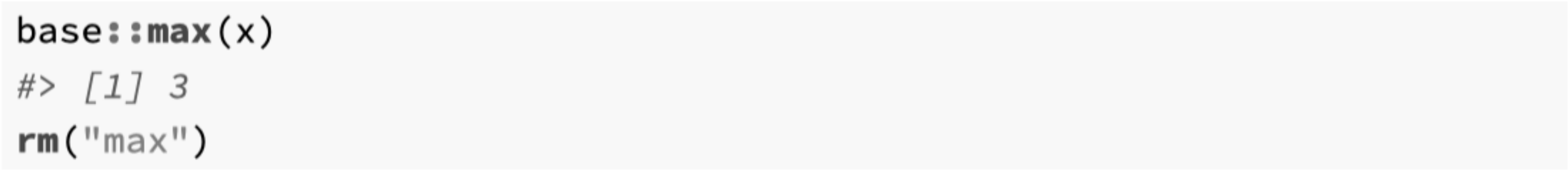
```
f1 <- function(a,b){
  (a+b)*z
}
environment(f1)
#> <environment: R_GlobalEnv>
```



Notice the use of env::func call

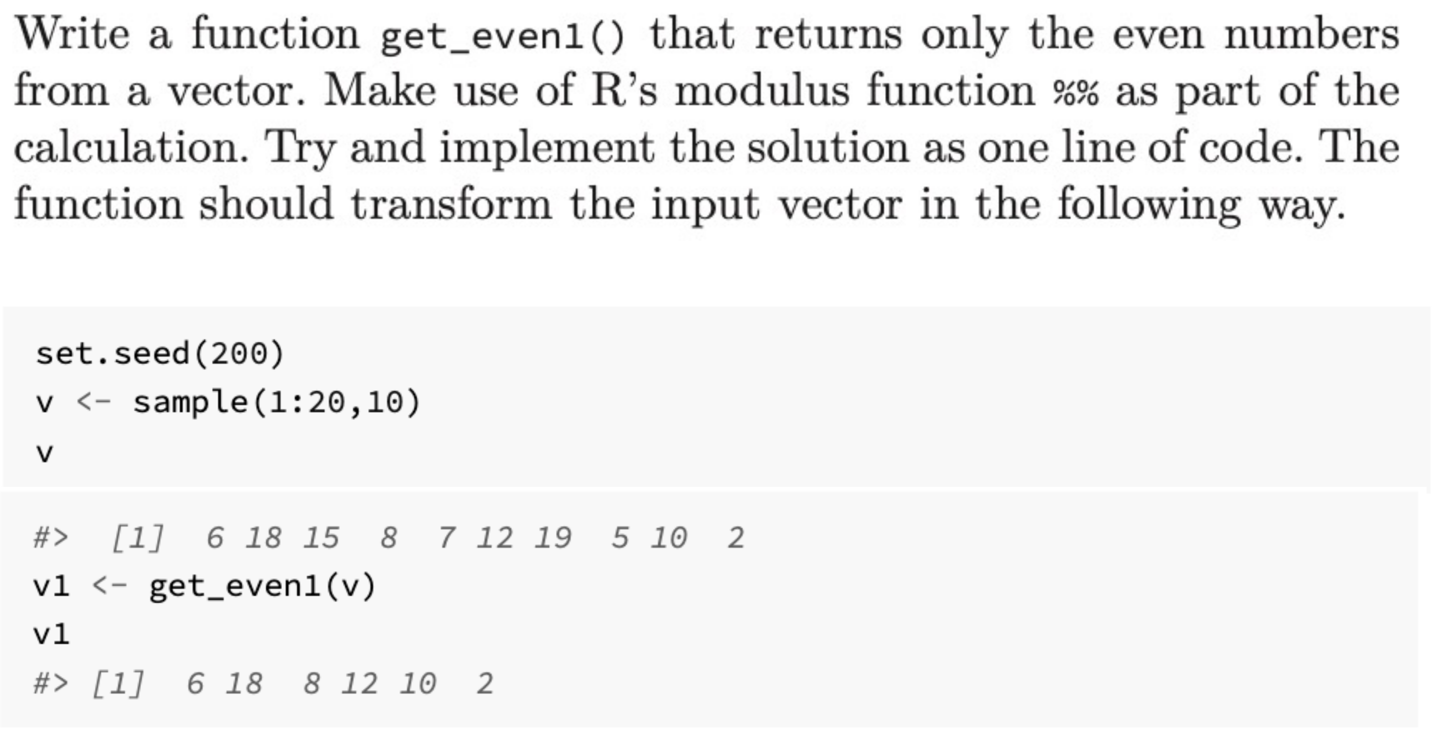


```r
max <- function(v){
  "Hello World"
}

ans <- max(x)
ans
#> [1] "Hello World"
```



```r
base::max(x)
#> [1] 3
rm("max")
```

Challenge 2.1



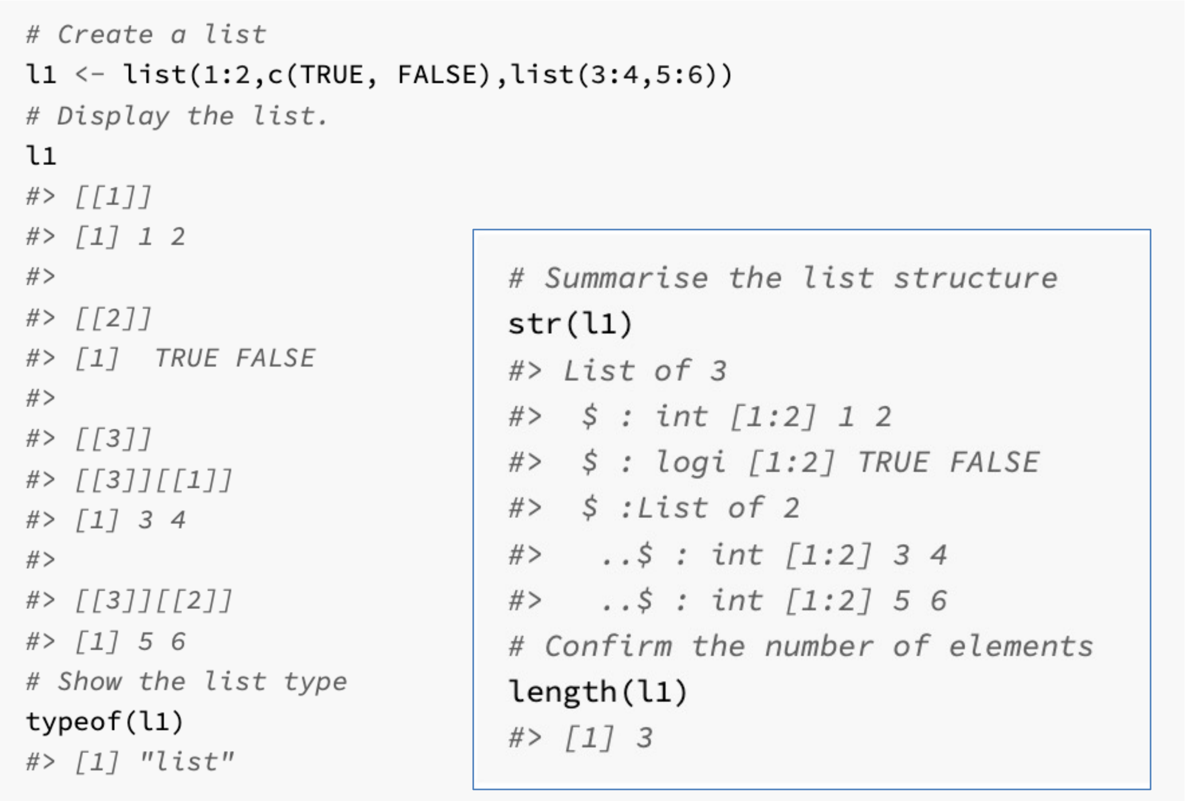
Write a function `get_even1()` that returns only the even numbers from a vector. Make use of R's modulus function `%%` as part of the calculation. Try and implement the solution as one line of code. The function should transform the input vector in the following way.

```
set.seed(200)
v <- sample(1:20,10)
v
```

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

Lists in R

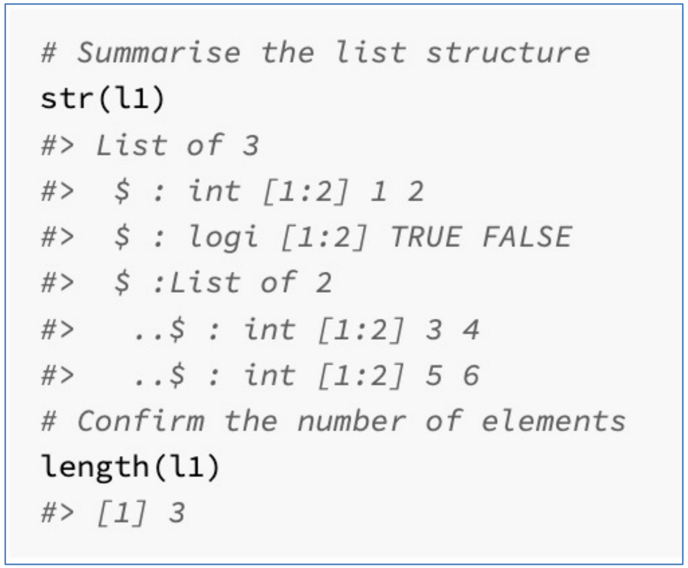
- A list is a vector that can contain different types, including a list.
- A list can be defined using the list() function
- This is similar to the c() function used to create atomic vectors.



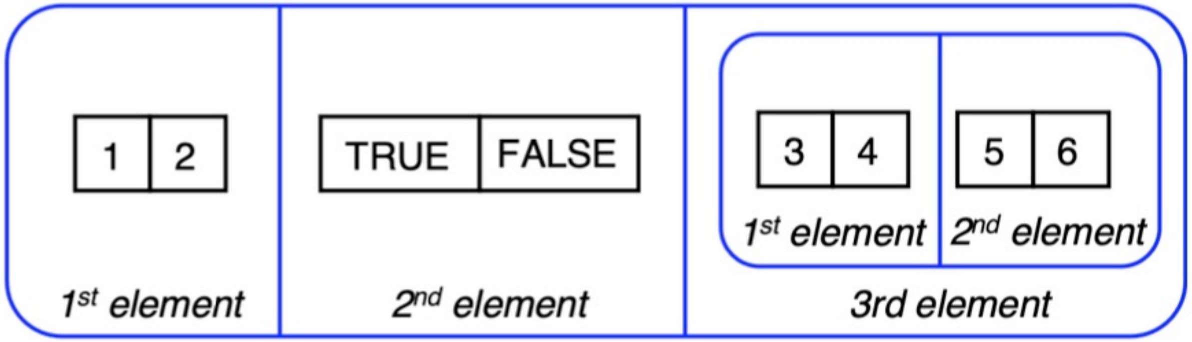
```
# Create a list
l1 <- list(1:2,c(TRUE, FALSE),list(3:4,5:6))
# Display the list.
l1
#> [[1]]
#> [1] 1 2
#>
#> [[2]]
#> [1]  TRUE FALSE
#>
#> [[3]]
#> [[3]][[1]]
#> [1] 3 4
#>
#> [[3]][[2]]
#> [1] 5 6
# Show the list type
typeof(l1)
#> [1] "list"
```

```
# Summarise the list structure
str(l1)
#> List of 3
#>  $ : int [1:2] 1 2
#>  $ : logi [1:2] TRUE FALSE
#>  $ :List of 2
#>   ..$ : int [1:2] 3 4
#>   ..$ : int [1:2] 5 6
# Confirm the number of elements
length(l1)
#> [1] 3
```

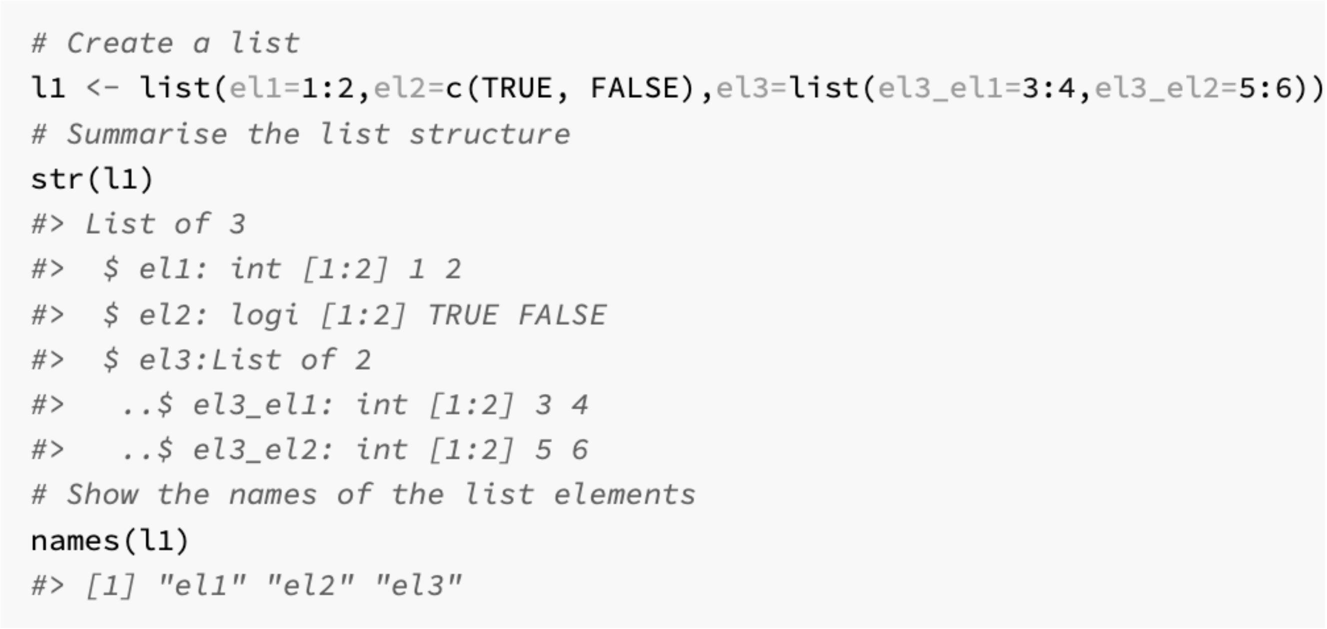
Visualising a list



```
# Summarise the list structure
str(l1)
#> List of 3
#>  $ : int [1:2] 1 2
#>  $ : logi [1:2] TRUE FALSE
#>  $ :List of 2
#>   ..$ : int [1:2] 3 4
#>   ..$ : int [1:2] 5 6
# Confirm the number of elements
length(l1)
#> [1] 3
```

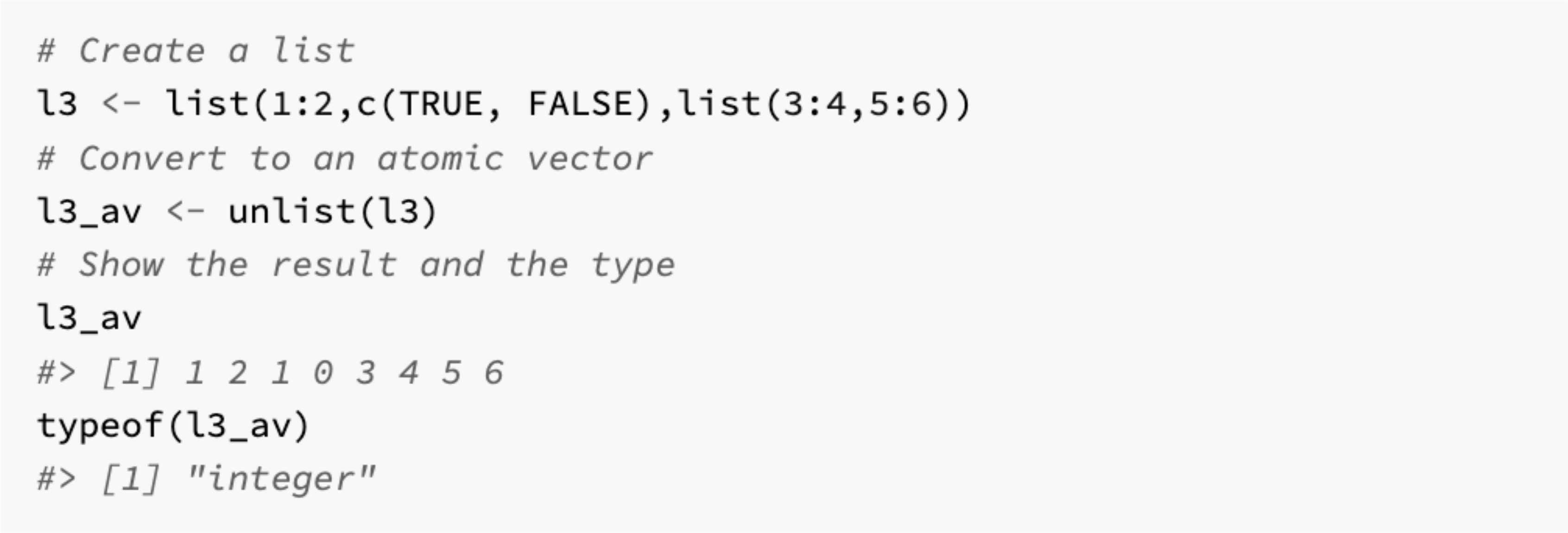


Naming list elements



```
# Create a list
l1 <- list(el1=1:2,el2=c(TRUE, FALSE),el3=list(el3_el1=3:4,el3_el2=5:6))
# Summarise the list structure
str(l1)
#> List of 3
#>  $ el1: int [1:2] 1 2
#>  $ el2: logi [1:2] TRUE FALSE
#>  $ el3:List of 2
#>   ..$ el3_el1: int [1:2] 3 4
#>   ..$ el3_el2: int [1:2] 5 6
# Show the names of the list elements
names(l1)
#> [1] "el1" "el2" "el3"
```

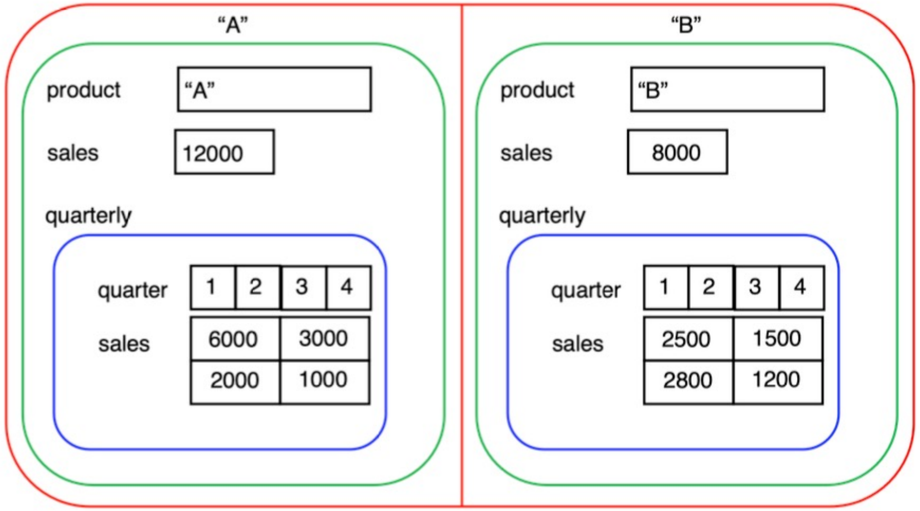
Conversion to an atomic vector



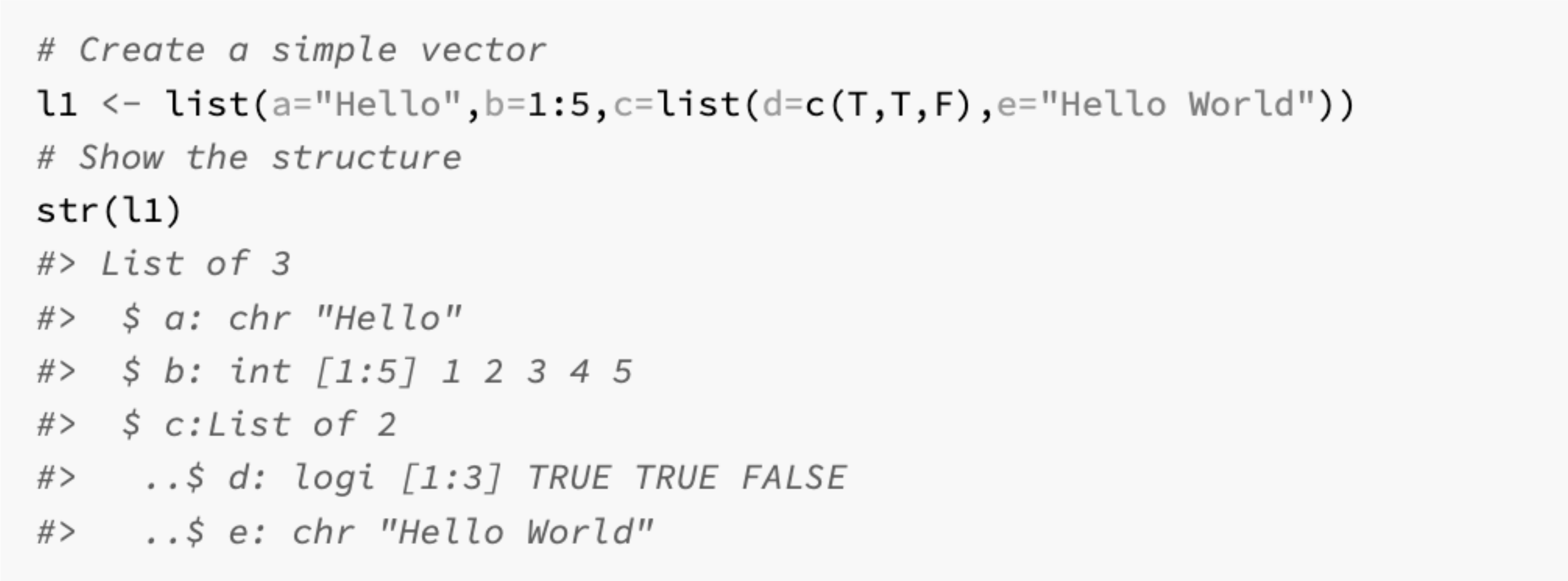
```
# Create a list
l3 <- list(1:2,c(TRUE, FALSE),list(3:4,5:6))
# Convert to an atomic vector
l3_av <- unlist(l3)
# Show the result and the type
l3_av
#> [1] 1 2 1 0 3 4 5 6
typeof(l3_av)
#> [1] "integer"
```

Subsetting lists

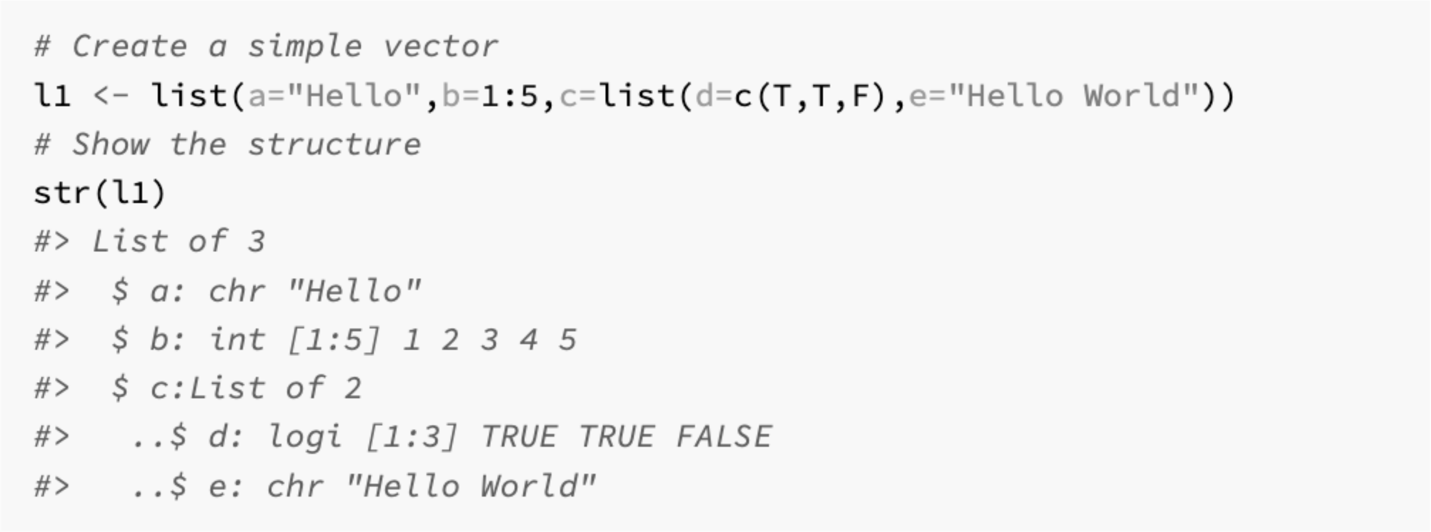
- The single square bracket [will always return a list, an dis similar to what we used for atomic vectors
- The double square bracket [[will return the contents of the list at a specified location
- The tag $ operator is a convenient way to extract the contents of a list (similar to [[)



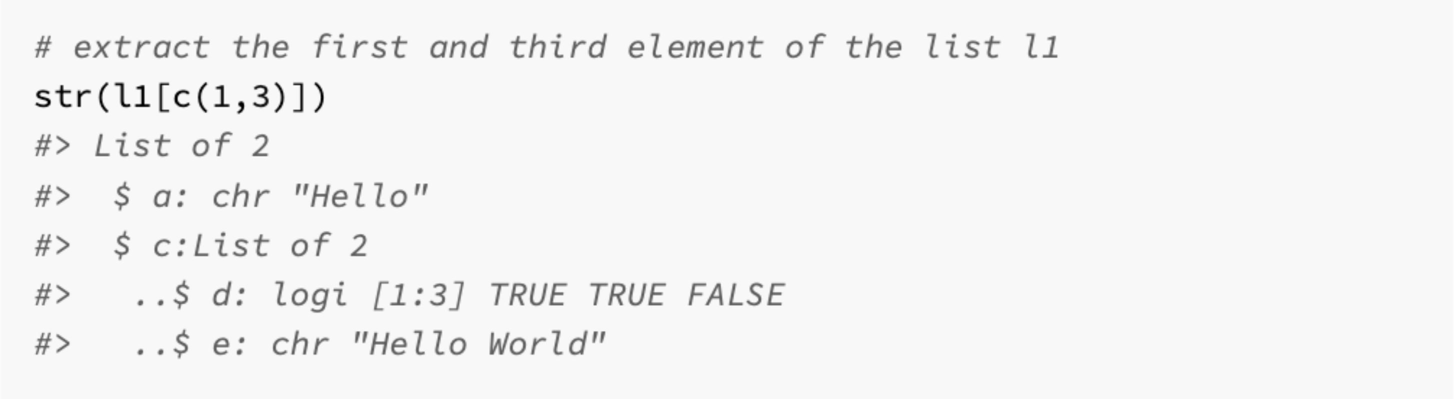
Exploring subsetting – a simple example



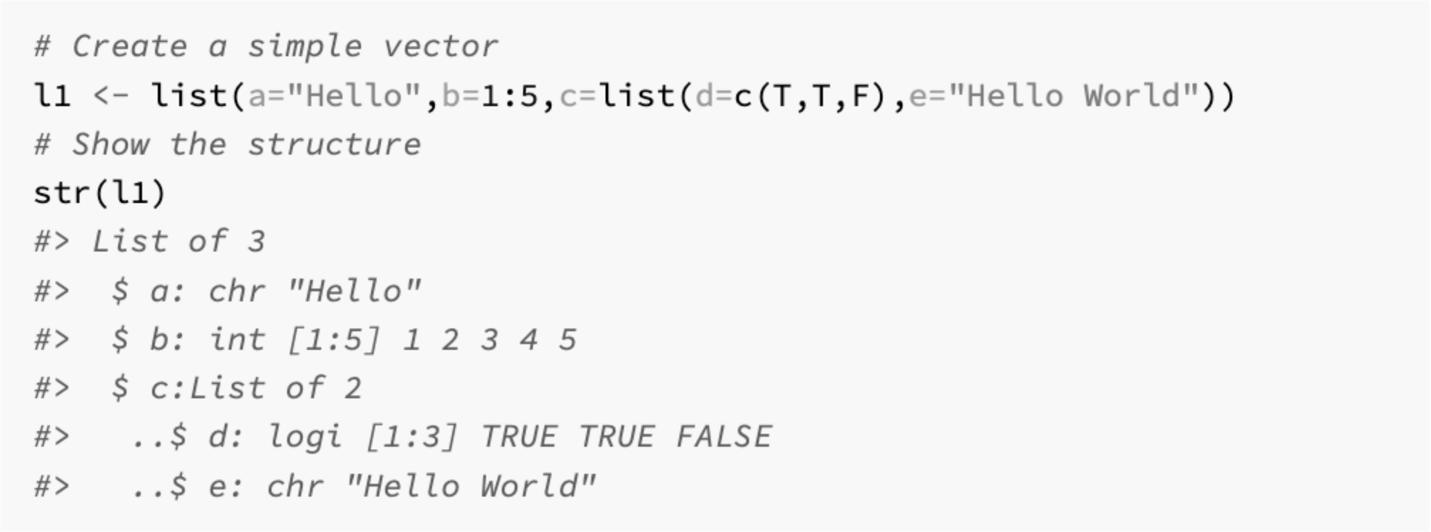
```
# Create a simple vector
l1 <- list(a="Hello",b=1:5,c=list(d=c(T,T,F),e="Hello World"))
# Show the structure
str(l1)
#> List of 3
#>  $ a: chr "Hello"
#>  $ b: int [1:5] 1 2 3 4 5
#>  $ c:List of 2
#>   ..$ d: logi [1:3] TRUE TRUE FALSE
#>   ..$ e: chr "Hello World"
```



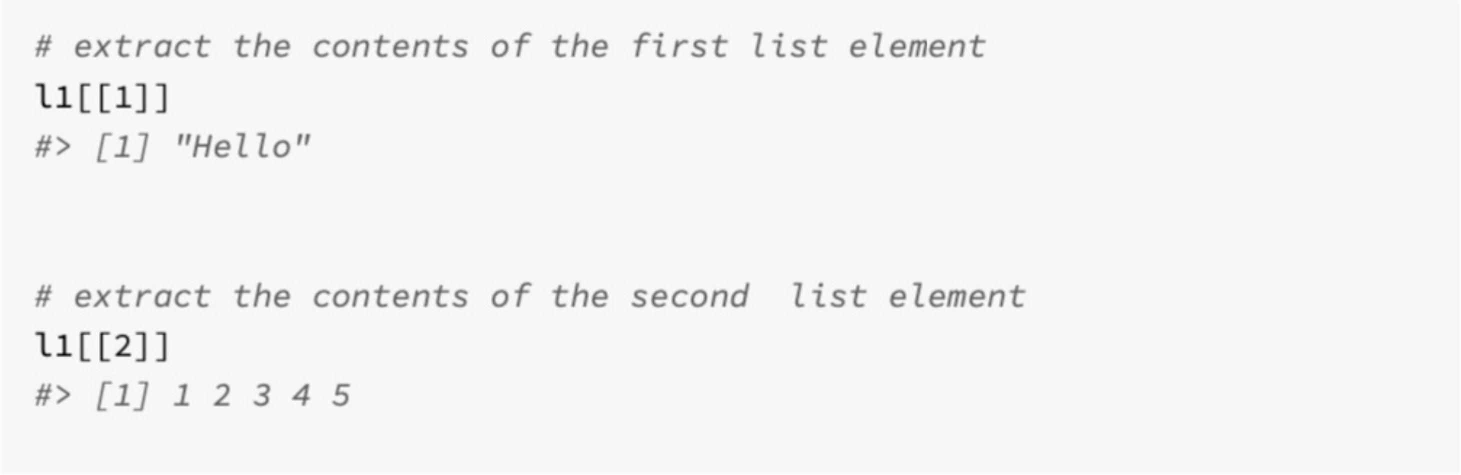
```
# Create a simple vector
l1 <- list(a="Hello",b=1:5,c=list(d=c(T,T,F),e="Hello World"))
# Show the structure
str(l1)
#> List of 3
#>  $ a: chr "Hello"
#>  $ b: int [1:5] 1 2 3 4 5
#>  $ c:List of 2
#>   ..$ d: logi [1:3] TRUE TRUE FALSE
#>   ..$ e: chr "Hello World"
```



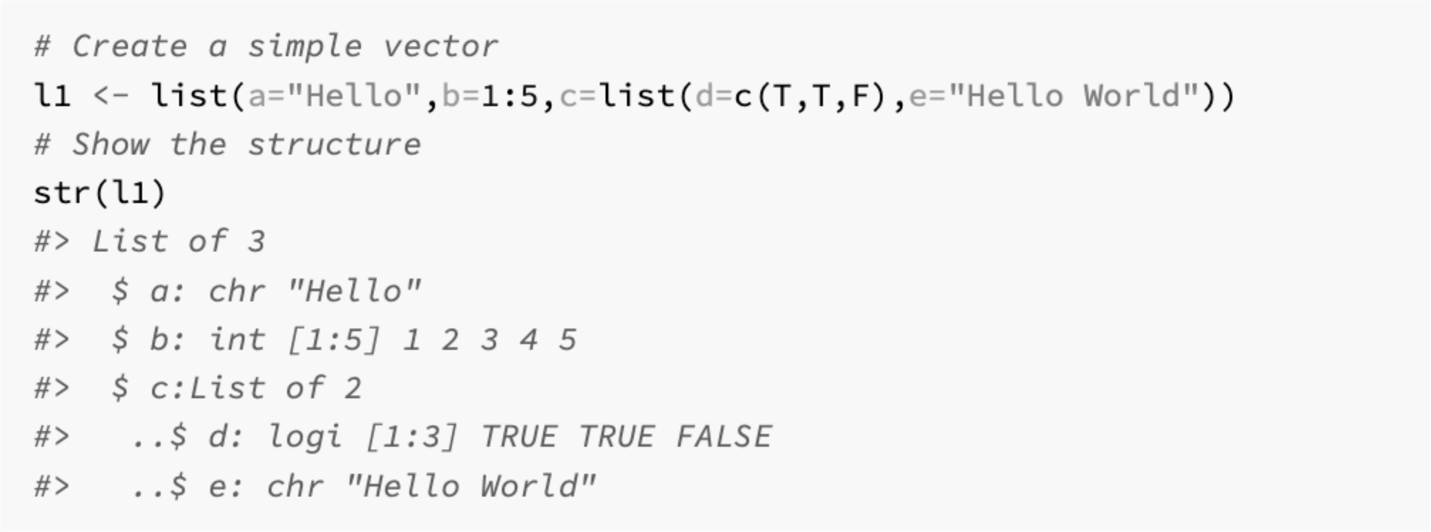
```
# extract the first and third element of the list l1
str(l1[c(1,3)])
#> List of 2
#>  $ a: chr "Hello"
#>  $ c:List of 2
#>   ..$ d: logi [1:3] TRUE TRUE FALSE
#>   ..$ e: chr "Hello World"
```



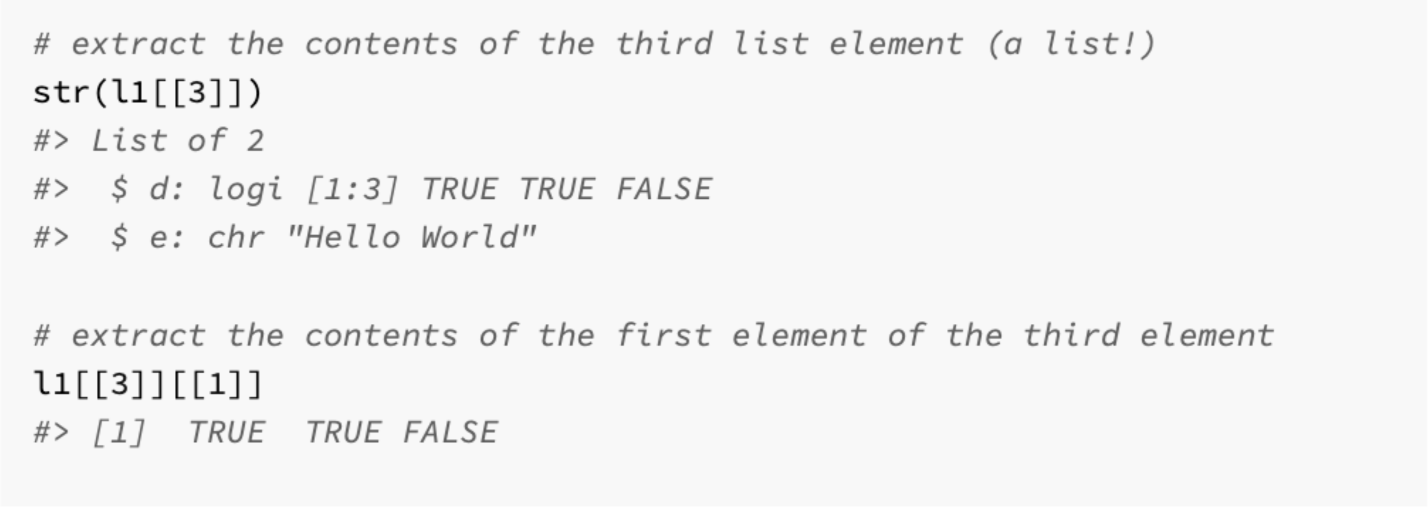
```
# Create a simple vector
l1 <- list(a="Hello",b=1:5,c=list(d=c(T,T,F),e="Hello World"))
# Show the structure
str(l1)
#> List of 3
#>  $ a: chr "Hello"
#>  $ b: int [1:5] 1 2 3 4 5
#>  $ c:List of 2
#>   ..$ d: logi [1:3] TRUE TRUE FALSE
#>   ..$ e: chr "Hello World"
```



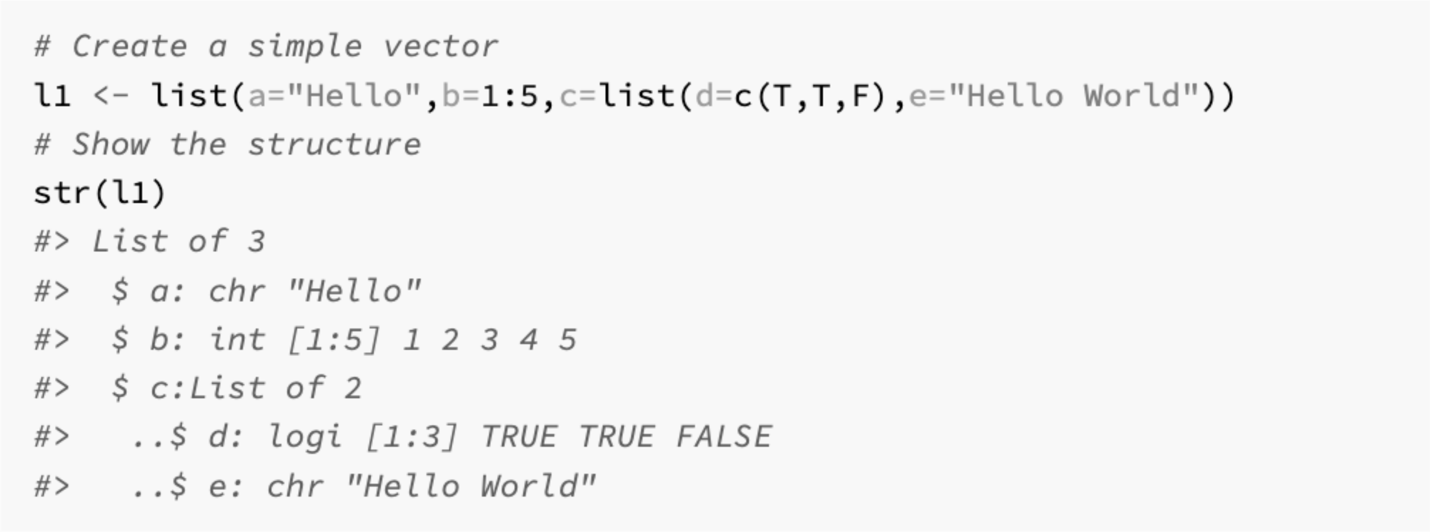
```
# extract the contents of the first list element
l1[[1]]
#> [1] "Hello"


# extract the contents of the second  list element
l1[[2]]
#> [1] 1 2 3 4 5
```



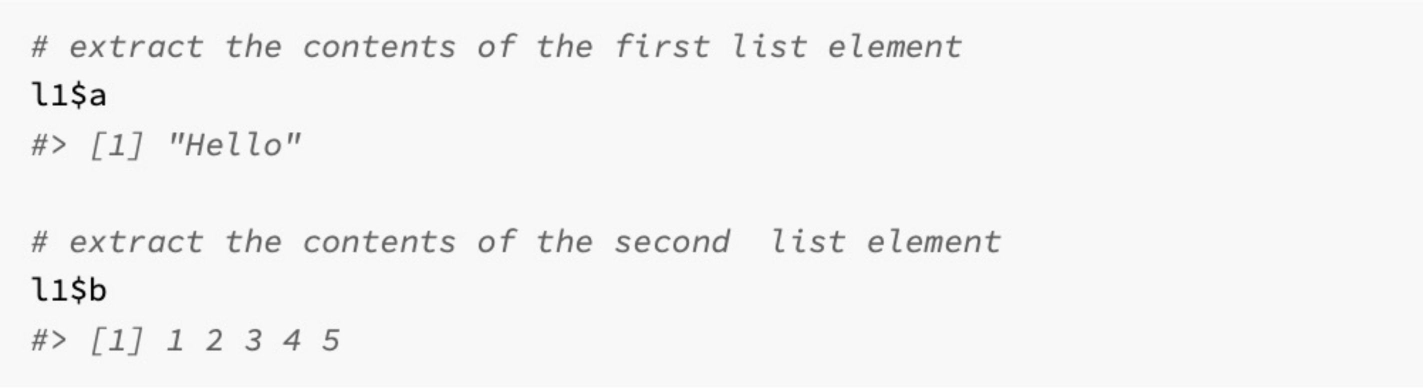
```r
# Create a simple vector
l1 <- list(a="Hello",b=1:5,c=list(d=c(T,T,F),e="Hello World"))
# Show the structure
str(l1)
#> List of 3
#>  $ a: chr "Hello"
#>  $ b: int [1:5] 1 2 3 4 5
#>  $ c:List of 2
#>   ..$ d: logi [1:3] TRUE TRUE FALSE
#>   ..$ e: chr "Hello World"
```



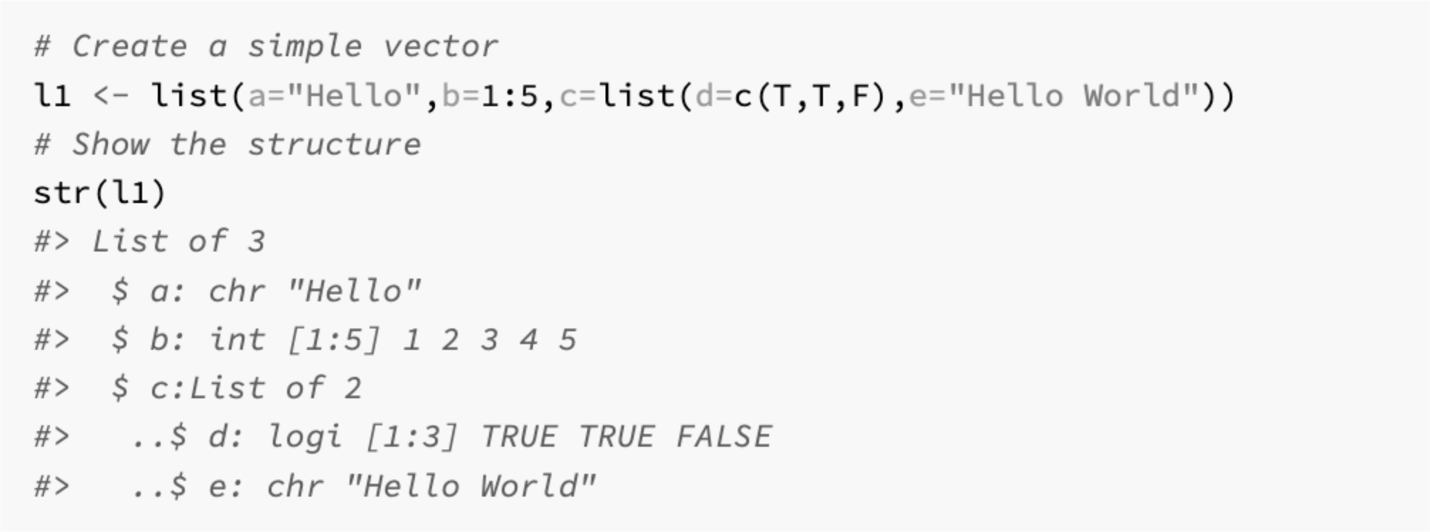
```r
# extract the contents of the third list element (a list!)
str(l1[[3]])
#> List of 2
#>  $ d: logi [1:3] TRUE TRUE FALSE
#>  $ e: chr "Hello World"

# extract the contents of the first element of the third element
l1[[3]][[1]]
#> [1]  TRUE  TRUE FALSE
```



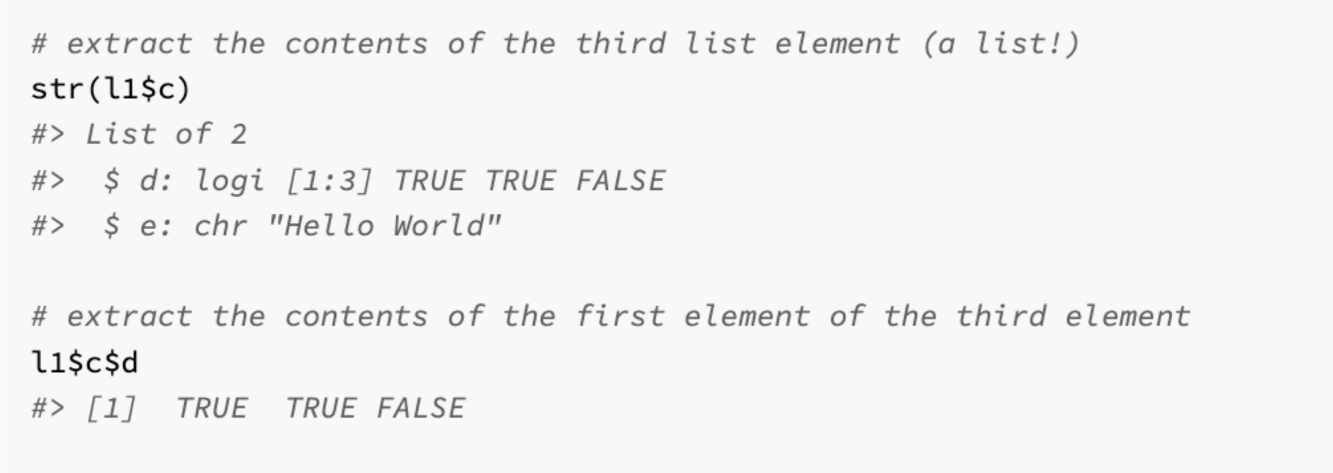
```r
# Create a simple vector
l1 <- list(a="Hello",b=1:5,c=list(d=c(T,T,F),e="Hello World"))
# Show the structure
str(l1)
#> List of 3
#>  $ a: chr "Hello"
#>  $ b: int [1:5] 1 2 3 4 5
#>  $ c:List of 2
#>   ..$ d: logi [1:3] TRUE TRUE FALSE
#>   ..$ e: chr "Hello World"
```



```r
# extract the contents of the first list element
l1$a
#> [1] "Hello"

# extract the contents of the second  list element
l1$b
#> [1] 1 2 3 4 5
```

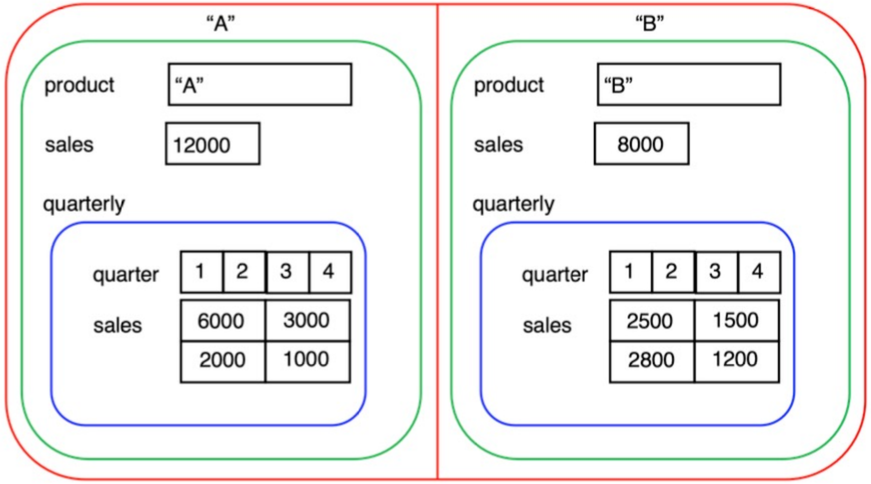


```
# Create a simple vector
l1 <- list(a="Hello",b=1:5,c=list(d=c(T,T,F),e="Hello World"))
# Show the structure
str(l1)
#> List of 3
#>  $ a: chr "Hello"
#>  $ b: int [1:5] 1 2 3 4 5
#>  $ c:List of 2
#>   ..$ d: logi [1:3] TRUE TRUE FALSE
#>   ..$ e: chr "Hello World"
```



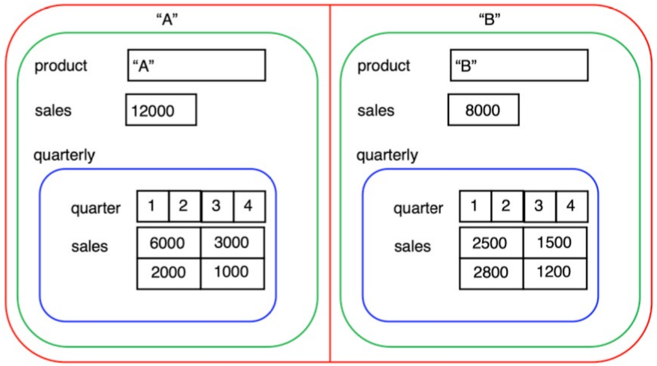
```
# extract the contents of the third list element (a list!)
str(l1$c)
#> List of 2
#>  $ d: logi [1:3] TRUE TRUE FALSE
#>  $ e: chr "Hello World"

# extract the contents of the first element of the third element
l1$c$d
#> [1]  TRUE  TRUE FALSE
```

Exploring another list…

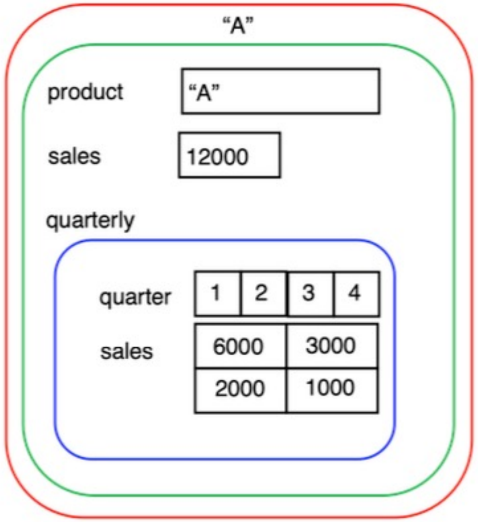


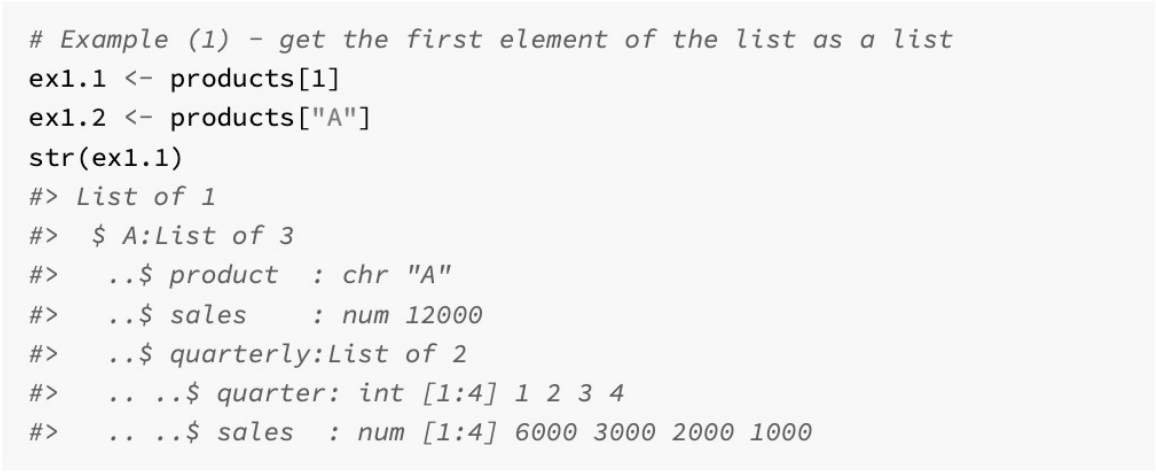
```r
# A small products database. Main list has two products
products <- list(A=list(product="A",sales=12000,
                        quarterly=list(quarter=1:4,
                                       sales=c(6000,3000,2000,1000))),
                 B=list(product="B",sales=8000,
                        quarterly=list(quarter=1:4,
                                       sales=c(2500,1500,2800,1200))))
```

```
str(products)
#> List of 2
#>  $ A:List of 3
#>   ..$ product  : chr "A"
#>   ..$ sales    : num 12000
#>   ..$ quarterly:List of 2
#>   .. ..$ quarter: int [1:4] 1 2 3 4
#>   .. ..$ sales  : num [1:4] 6000 3000 2000 1000
#>  $ B:List of 3
#>   ..$ product  : chr "B"
#>   ..$ sales    : num 8000
#>   ..$ quarterly:List of 2
#>   .. ..$ quarter: int [1:4] 1 2 3 4
#>   .. ..$ sales  : num [1:4] 2500 1500 2800 1200
```

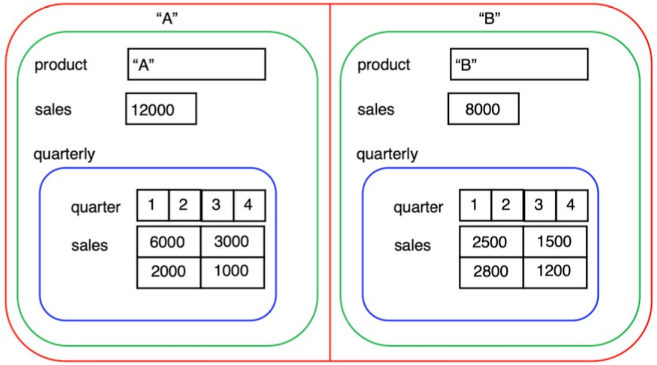


(1) products[1] *or* products["A"]





```
# Example (1) - get the first element of the list as a list
ex1.1 <- products[1]
ex1.2 <- products["A"]
str(ex1.1)
#> List of 1
#>  $ A:List of 3
#>   ..$ product  : chr "A"
#>   ..$ sales    : num 12000
#>   ..$ quarterly:List of 2
#>   .. ..$ quarter: int [1:4] 1 2 3 4
#>   .. ..$ sales  : num [1:4] 6000 3000 2000 1000
```



"A"

product "A"

sales 12000

quarterly

quarter 1 2 3 4

sales 6000 3000 2000 1000

"B"

product "B"

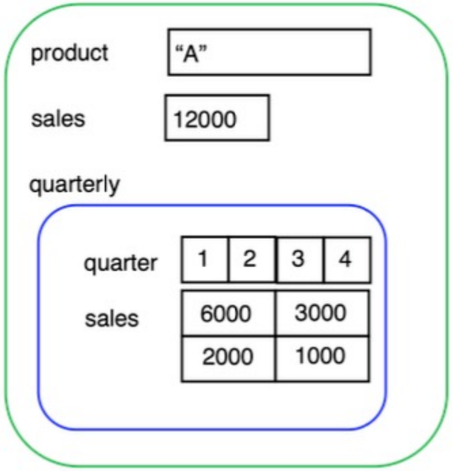
sales 8000

quarterly

quarter 1 2 3 4

sales 2500 1500 2800 1200

(2) products[[1]] *or* products[["A"]] *or* products$A



product "A"

sales 12000

quarterly

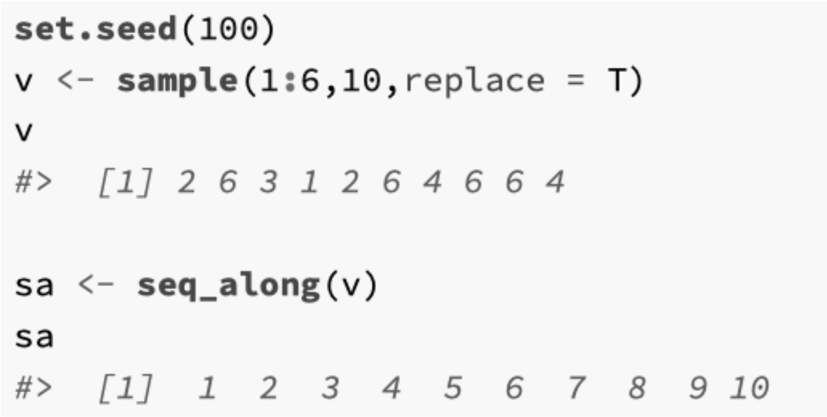
quarter 1 2 3 4

sales 6000 3000 2000 1000

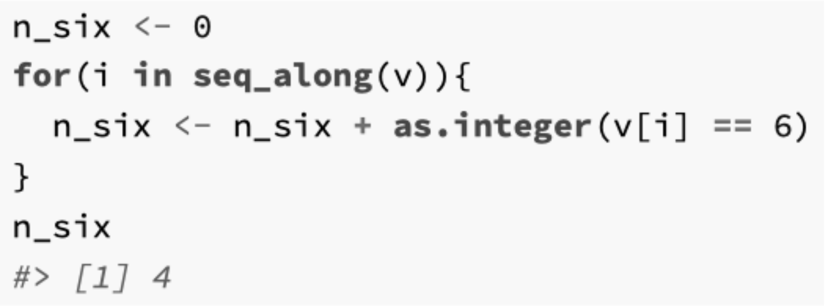
```
# Example (2) - get the contents of the first list element
ex2.1 <- products[[1]]
ex2.2 <- products[["A"]]
ex2.3 <- products$A
str(ex2.1)
#> List of 3
#>  $ product  : chr "A"
#>  $ sales    : num 12000
#>  $ quarterly:List of 2
#>   ..$ quarter: int [1:4] 1 2 3 4
#>   ..$ sales  : num [1:4] 6000 3000 2000 1000
```

Iteration

- Iteration is fundamental to all programming languages, and R is no exception.
- There are a number of basic looping structures than can be used in R, including the for loop. The general structure is ‘for(var in seq)expr, where:
  - var is a name for a variable that will change its value for each loop iteration
  - seq is an expression that evaluates to a vector
- expr which is an expression, which can be either a simple expression, or a compound expression of the form {expr1; expr2}, which is effectively a number of lines of code with two curly braces.
- A convenient method to iterate over a vector, is to use the function seq_along() which returns the indices of a vector. For example, consider the vector v below, which contains a simulation of ten dice rolls.



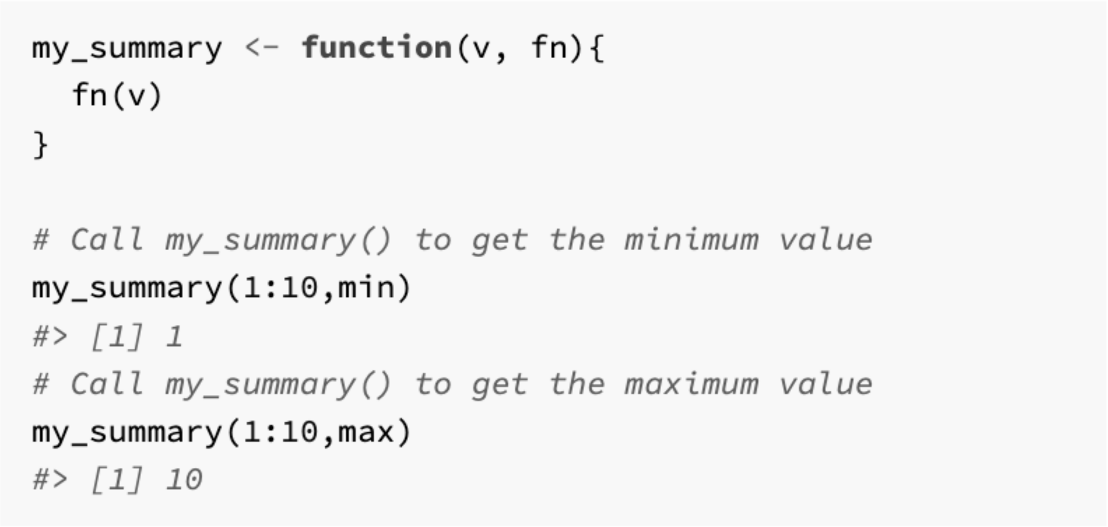
```
set.seed(100)
v <- sample(1:6,10,replace = T)
v
#>  [1] 2 6 3 1 2 6 4 6 6 4

sa <- seq_along(v)
sa
#>  [1]  1  2  3  4  5  6  7  8  9 10
```



```
n_six <- 0
for(i in seq_along(v)){
  n_six <- n_six + as.integer(v[i] == 6)
}
n_six
#> [1] 4
```

Functionals

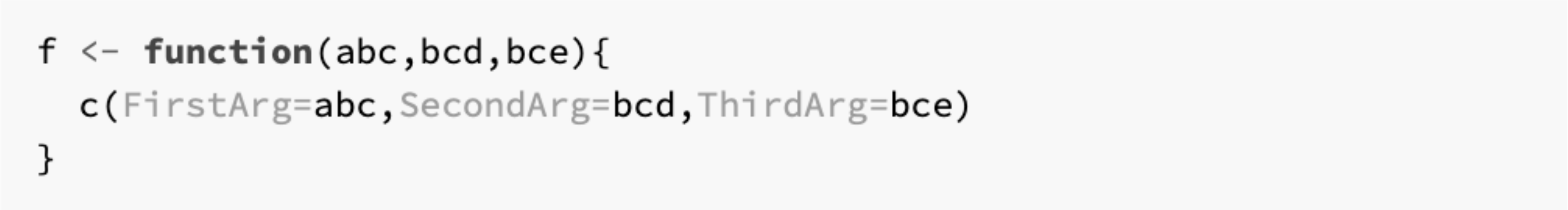
- A function can accept another function as an argument
- These functions are known as functionals and are a key part of R



```
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
```

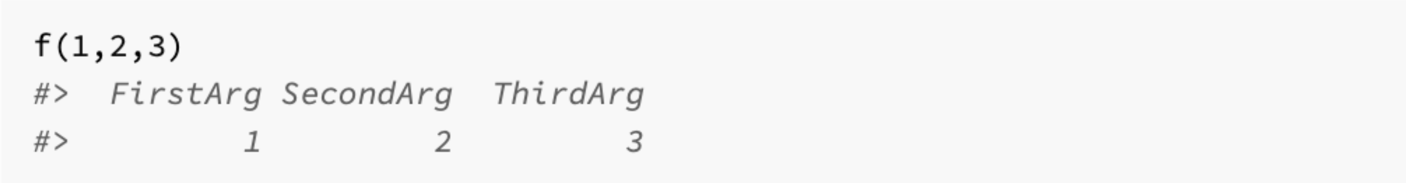
Arguments

- 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
- However, arguments can also be passed in by complete name or partial name, and arguments can also have default values

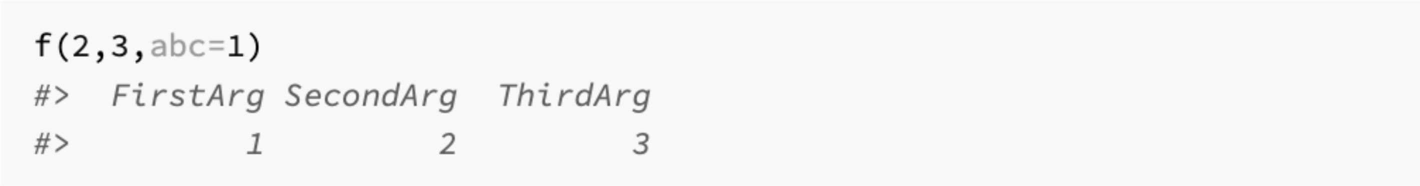


```
f <- function(abc,bcd,bce){
  c(FirstArg=abc,SecondArg=bcd,ThirdArg=bce)
}
```

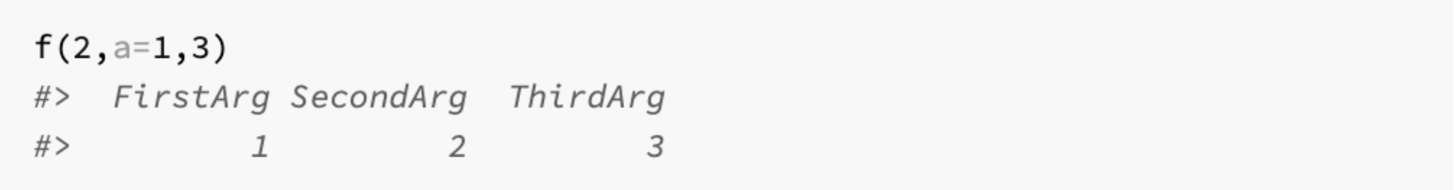
Flexibility in calls



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

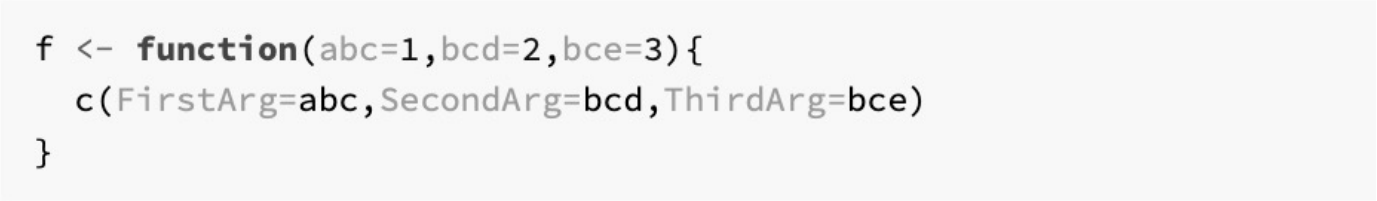


```
f(2,3,abc=1)
#>  FirstArg SecondArg  ThirdArg
#>         1         2         3
```

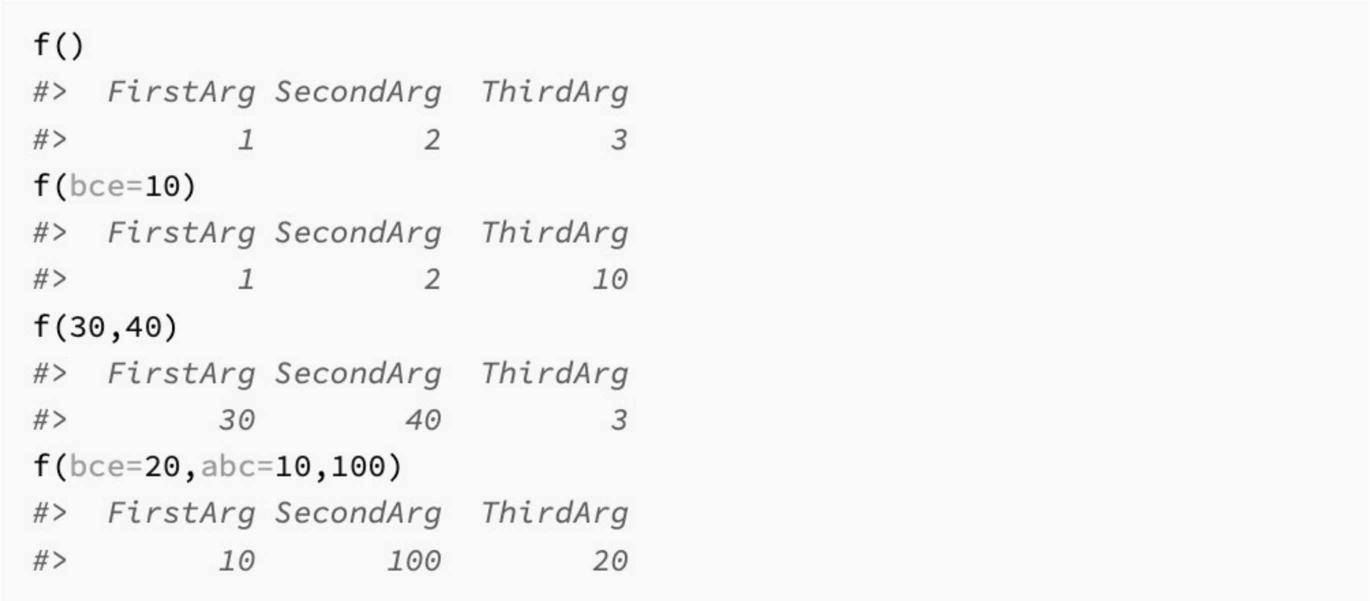


```
f(2,a=1,3)
#>  FirstArg SecondArg  ThirdArg
#>         1         2         3
```

Setting default values



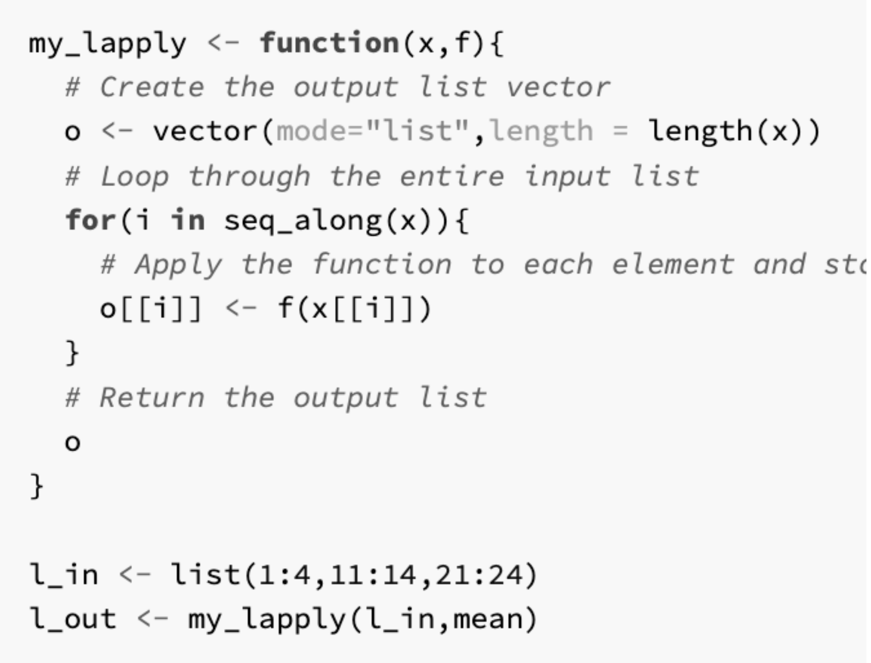
```r
f <- function(abc=1,bcd=2,bce=3){
  c(FirstArg=abc,SecondArg=bcd,ThirdArg=bce)
}
```



```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
```

The apply family of functionals

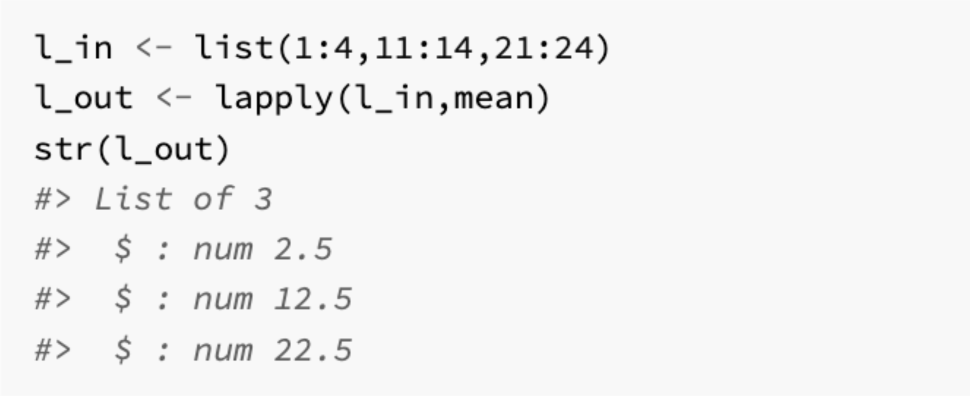
- An important aspect of programming with R, which is the use of functionals
- They take data and functions as part of their input, and use that function to process data.
- In many cases, these functions can be used instead of loops to iterate over data and return a result



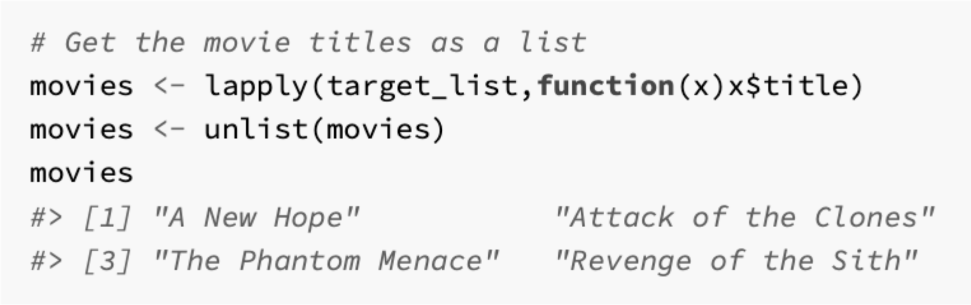
```r
my_lapply <- function(x,f){
  # Create the output list vector
  o <- vector(mode="list",length = length(x))
  # Loop through the entire input list
  for(i in seq_along(x)){
    # Apply the function to each element and st
    o[[i]] <- f(x[[i]])
  }
  # Return the output list
  o
}

l_in <- list(1:4,11:14,21:24)
l_out <- my_lapply(l_in,mean)
```

lapply(x,f)

- 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 function can be embedded (anonymous) – example from repurrrsive.



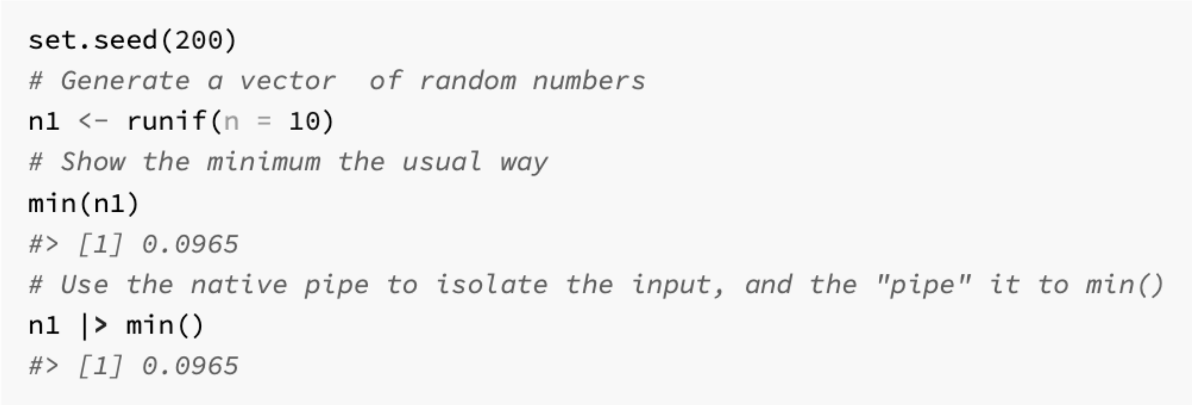
```r
l_in <- list(1:4,11:14,21:24)
l_out <- lapply(l_in,mean)
str(l_out)
#> List of 3
#>  $ : num 2.5
#>  $ : num 12.5
#>  $ : num 22.5
```



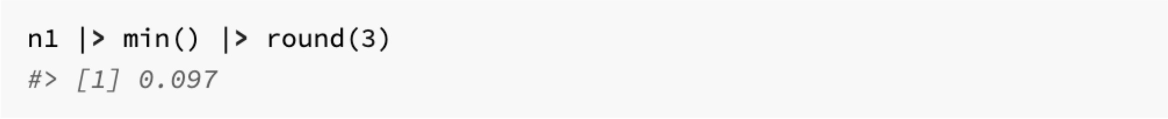
```r
# Get the movie titles as a list
movies <- lapply(target_list,function(x)x$title)
movies <- unlist(movies)
movies
#> [1] "A New Hope"           "Attack of the Clones"
#> [3] "The Phantom Menace"   "Revenge of the Sith"
```

Native Pipe Operator |>

- Allows you to chain a number of operations together, without having to assign intermediate variables
- Can construct a data pipeline
- The general format of the pipe operator is LHS |> RHS, where LHS is the first argument of the function defined on the RHS.

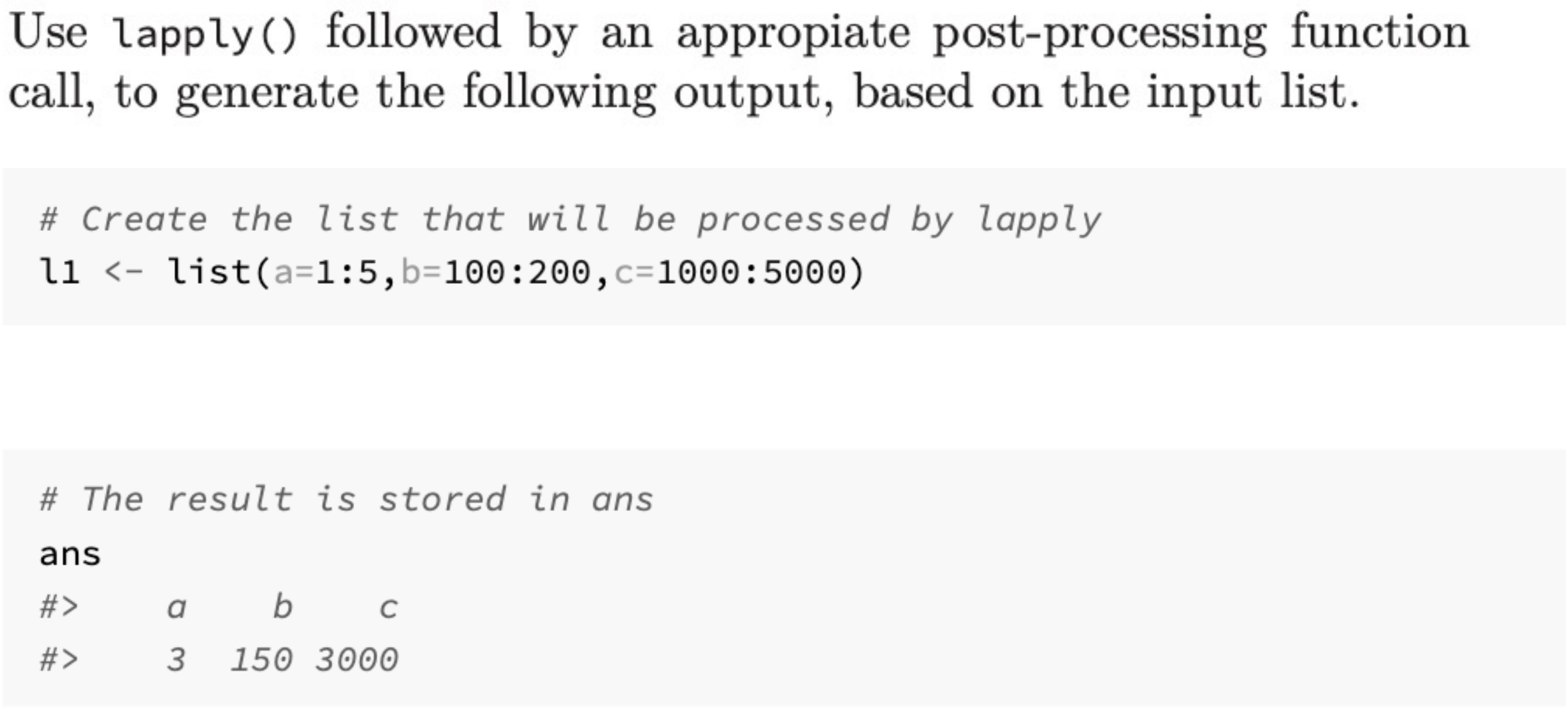


```
set.seed(200)
# Generate a vector  of random numbers
n1 <- runif(n = 10)
# Show the minimum the usual way
min(n1)
#> [1] 0.0965
# Use the native pipe to isolate the input, and the "pipe" it to min()
n1 |> min()
#> [1] 0.0965
```



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

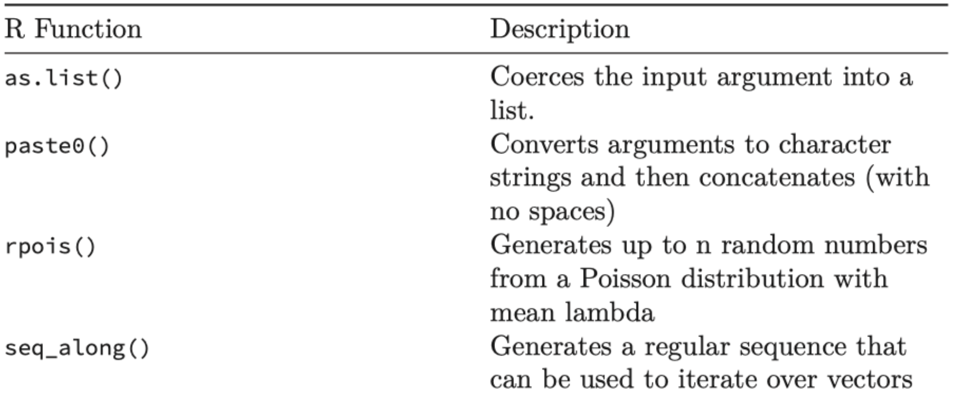
Challenge 2.2



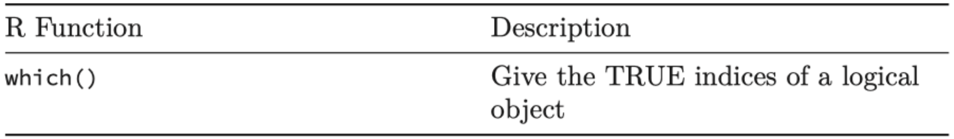
Use `lapply()` followed by an appropiate post-processing function call, to generate the following output, based on the input list.

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

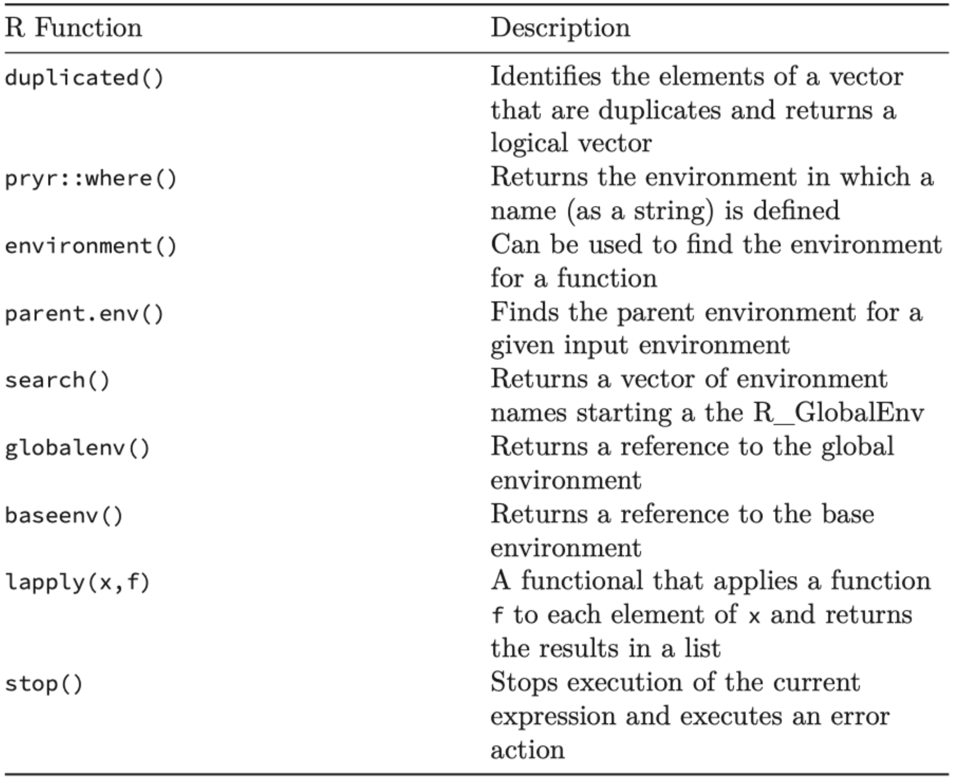
```
# The result is stored in ans
ans
#>    a    b    c
#>    3  150 3000
```



R Function	Description
`as.list()`	Coerces the input argument into a list.
`paste0()`	Converts arguments to character strings and then concatenates (with no spaces)
`rpois()`	Generates up to n random numbers from a Poisson distribution with mean lambda
`seq_along()`	Generates a regular sequence that can be used to iterate over vectors



R Function	Description
`which()`	Give the TRUE indices of a logical object



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
`lapply(x,f)`	A functional that applies a function `f` to each element of `x` and returns the results in a list
`stop()`	Stops execution of the current expression and executes an error action