

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](#))

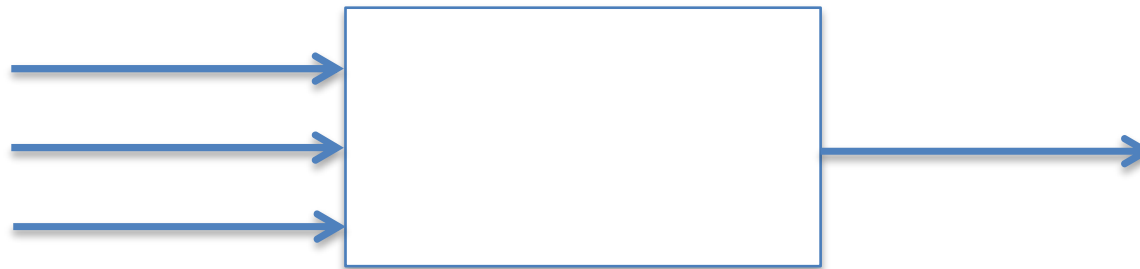
# Data Science for Operational Researchers using R

## 03 – Functions, Lists and Functionals

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

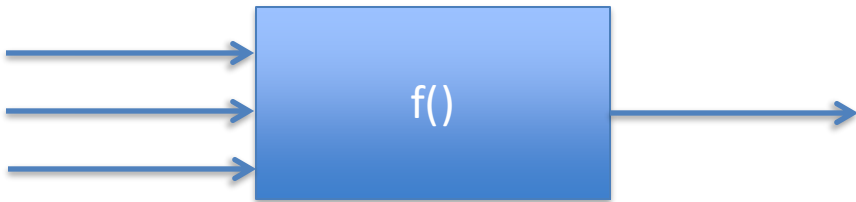
# (1) Functions

- 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

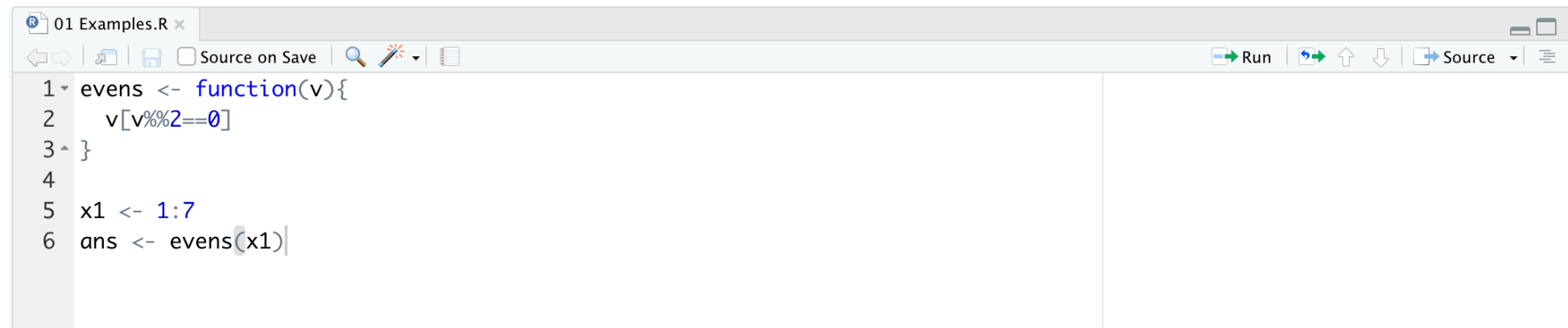
- Our first function will take in a vector of numbers, and return only those that are even.
- To do this, R's modulus operator `%%` is used, as this returns the remainder of two numbers, following their division
- We will focus on the data processing first, and then encapsulate this within a function

```
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
```



The screenshot shows the RStudio interface with a single script file named '01 Examples.R'. The code editor contains the following R code:

```
1 evens <- function(v){  
2   v[v%%2==0]  
3 }  
4  
5 x1 <- 1:7  
6 ans <- evens(x1)
```

The interface includes a toolbar with icons for navigation, saving, and running code. The 'Run' button is highlighted in green. The console area on the right is currently empty.

# 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

```
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  
}
```

```
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)]  
}
```



# Challenge 3.1

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

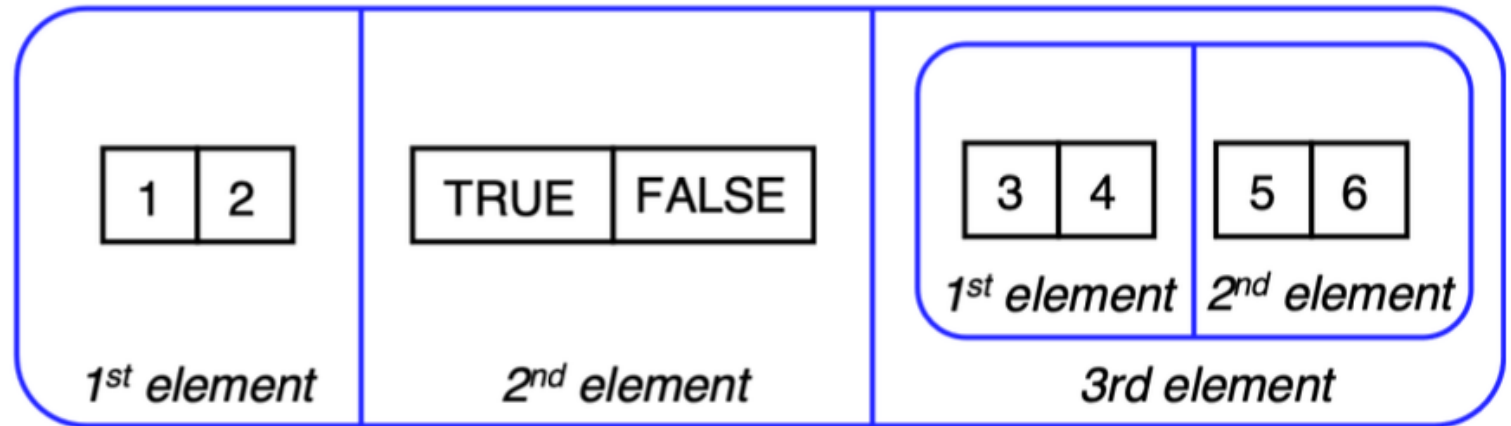
- A list is a vector that can contain different types, including a list.
- A list can be defined using the `list()` function
- This is similar to the `c()` function used to create atomic vectors.

```
# 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

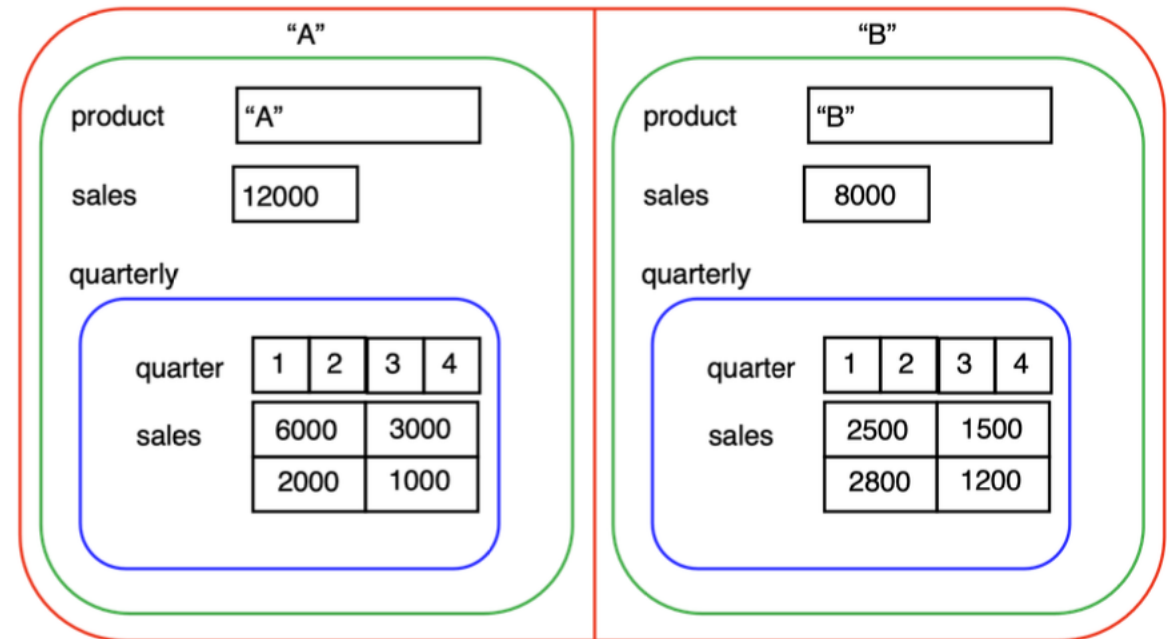
```
# Create a list
l1 <- list(el1=1:2,el2=c(TRUE, FALSE),el3=list(el3_el1=3:4,el3_el2=5:6))
# Summarise the list structure
str(l1)
#> List of 3
#> $ el1: int [1:2] 1 2
#> $ el2: logi [1:2] TRUE FALSE
#> $ el3:List of 2
#> ..$ el3_el1: int [1:2] 3 4
#> ..$ el3_el2: int [1:2] 5 6
# Show the names of the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

```
# 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[[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
```

```
# 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$a
#> [1] "Hello"

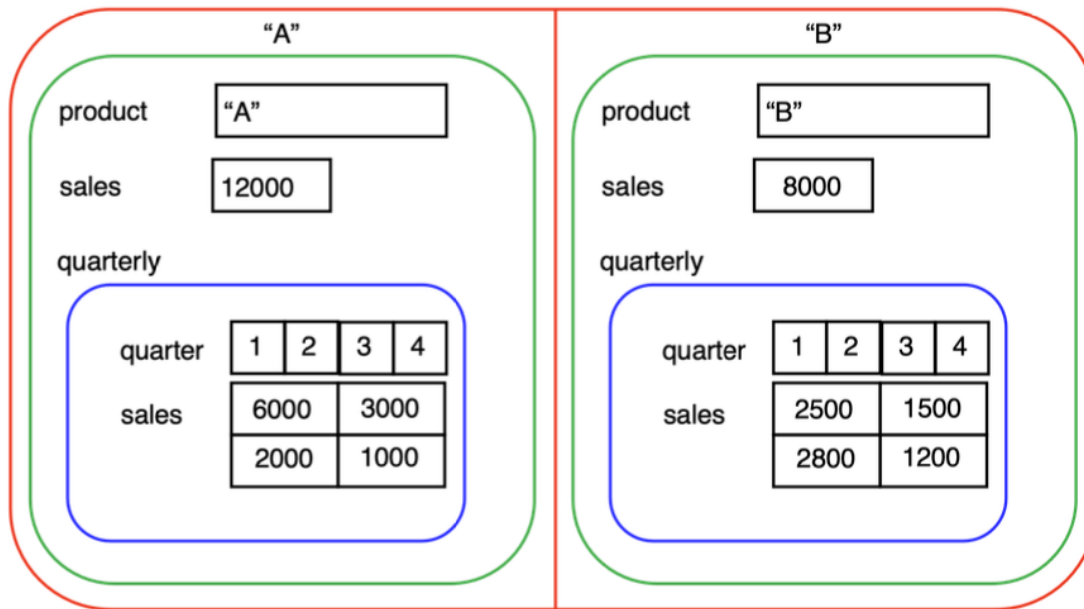
# extract the contents of the second list element
l1$b
#> [1] 1 2 3 4 5
```

```
# 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
```

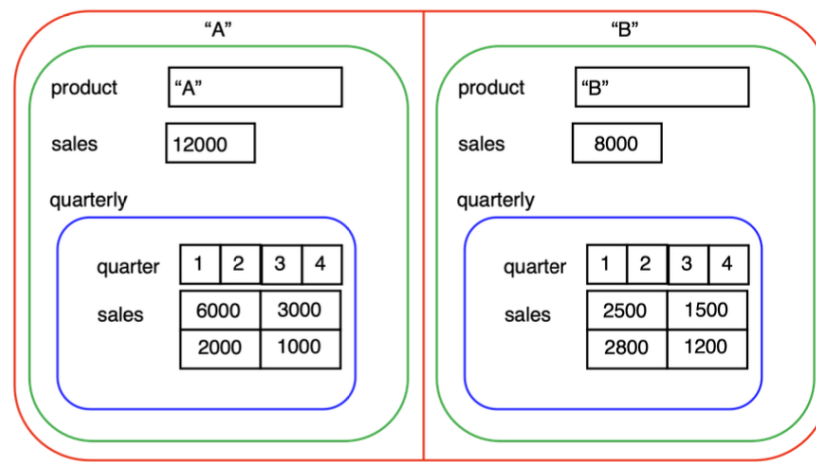
# Exploring another list...



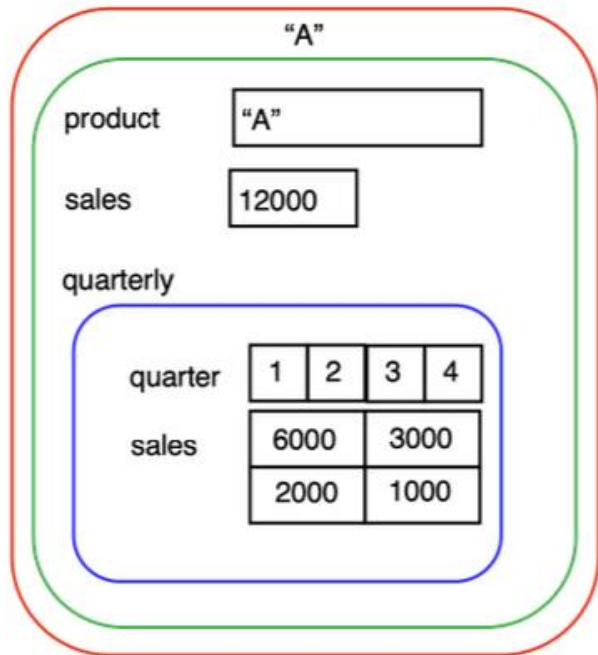
# A small products database. Main list has two products

```
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))))
```

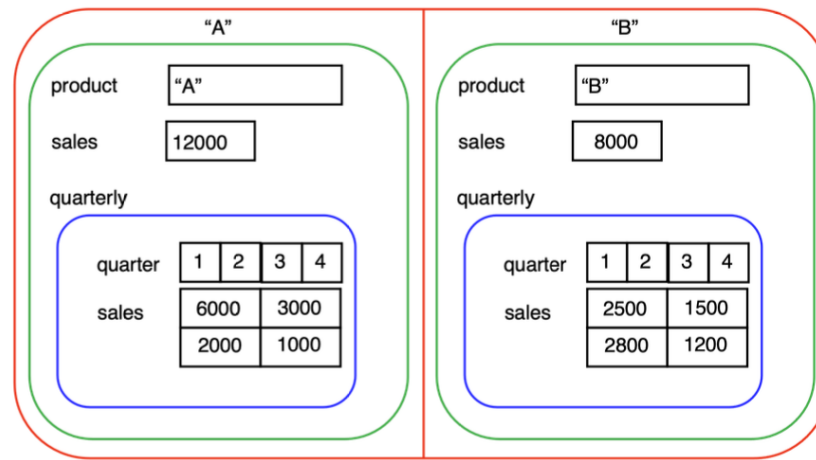
```
str(products)  
#> List of 2  
#> $ A:List of 3  
#> ..$ product : chr "A"  
#> ..$ sales    : num 12000  
#> ..$ quarterly:List of 2  
#> .. ..$ quarter: int [1:4] 1 2 3 4  
#> .. ..$ sales  : num [1:4] 6000 3000 2000 1000  
#> $ B:List of 3  
#> ..$ product  : chr "B"  
#> ..$ sales    : num 8000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



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



```
# 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
```

# 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

```
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
```

# 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

```
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 the result  
    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](#).

```
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
```

```
# 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 3.2

Use `lapply()` followed by an appropriate 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
```