2. Base R - Subsetting

CT5102 - J. Duggan (University of Galway)

Subsetting

R's subsetting operators are fast and powerful. Mastering them allows you to succinctly perform complex operations in a way that few other languages can match.
— Hadley Wickham "Advanced R"

Overview

- Subsetting operations allow you to process data stored in atomic vectors and lists
- R provides a range of flexible approaches that can be used to subset data.
- Learning outcomes:
  - The four main ways to subset a vector, namely, by positive integer, the minus sign, logical vectors, and vector element names.
  - The role of the `[[` operator for processing lists, and how to distinguish this from the `[` operator.
  - The use of the `$` operator
  - How to use the `for` loop structure to iterate through a list
  - How to use the `if` statement when processing lists, and how that statement differe from the `ifelse()` function covered in Chapter Chapter @ref(ch2).
  - Additional R functions that allow you to process vectors
  - How to solve all three test exercises.

Atomic Vectors

- Let's model the daily number of customers arriving at a restaurant as a *Poisson distribution*, with a mean (λ) of one hundred customers per day (λ=100).
- The Poisson distribution describes a discrete random variable, and the mean and variance are both equal to λ
- In R, the `rpois()` function can be used to generate random numbers from a Poisson distribution, with mean (λ).

```
# set the seed to ensure replication
set.seed(111)
# Generate the count data, assume a Poisson distribution
customers <- rpois(n = 10, lambda = 100)
names(customers) <- paste0("D",1:10)
customers
```

```
##  D1  D2  D3  D4  D5  D6  D7  D8  D9 D10
## 102  96  97  98 101  85  98 118 102  94
```

Subsetting vectors - Positive Integers

- For vectors in R, the operator `[` is used to subset vectors
- Positive integers will subset atomic vector elements at given locations
- To extract the n^{th} item from a vector `x` the term `x[n]` is used
- This can also apply to a sequence, starting at `n` and finising at `m` can be extracted from the vector `x` as `x[n:m]`
- Indices can also be generated using the combine function `c()`, which is then passed in to subset a vector.

```
customers[1]
```

```
##  D1
## 102
```

```
customers[1:5]
```

```
##  D1  D2  D3  D4  D5
## 102  96  97  98 101
```

Subsetting vectors - Negative Integers

- Negative integers, expressed as a vector, can be used to exclude elements from a vector
- One or more elements can be excluded

```
customers[-1]
```

```
##  D2  D3  D4  D5  D6  D7  D8  D9 D10
##  96  97  98 101  85  98 118 102  94
```

```
customers[-c(1,length(customers))]
```

```
##  D2  D3  D4  D5  D6  D7  D8  D9
##  96  97  98 101  85  98 118 102
```

```
customers[-(2:(length(customers)-1))]
```

```
##  D1 D10
## 102  94
```

Subsetting vectors - Logical Vectors

- Logical vectors can be used to subset a vector
- This allows for the use of relational and logical operators.
- when a logical vector is used to subset a vector, only the corresponding cells of the logical vector element that contain TRUE will be retained in the operation.

```
##    D1    D2    D3    D4    D5    D6    D7    D8    D9   D1
##  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALS
```

```
##  D1  D5  D8  D9
## 102 101 118 102
```

- The two statements can be combined into the one expression.

```
(customers[customers > 100])
```

```
##  D1  D5  D8  D9
## 102 101 118 102
```

Recycling

- A nice feature of subsetting with logical vectors is that the logical vector size does not have to equal the size of the target vector.
- When the length of the logical vector is less than the target vector, R will *recycle* the logical vector by repeating the sequence of values until all the target values have been subsetted.

```
# Subset every third element from the vector
customers[c(TRUE,FALSE,FALSE)]
```

```
##  D1  D4  D7 D10
## 102  98  98  94
```

Subsetting vectors - By Element Names

- If a vector has named elements - usually set via the function `names()`, then elements can subsetted through their name.
- This is convenient if you want to retrieve an element but do not necessarily want to know its exact indexed location.

```
customers
```

```
##  D1  D2  D3  D4  D5  D6  D7  D8  D9 D10
## 102  96  97  98 101  85  98 118 102  94
```

```
# Show the value from day 10
customers["D10"]
```

```
## D10
##  94
```

```
customers[c("D1","D10")]
```

```
##  D1 D10
```

Subsetting Lists

In a similar manner to our exploration of atomic vectors, we first generate a simulated manufacturing data for two products, A and B.

```
# 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))))
```

Exploring the list structure

```
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
```

Visualising the List

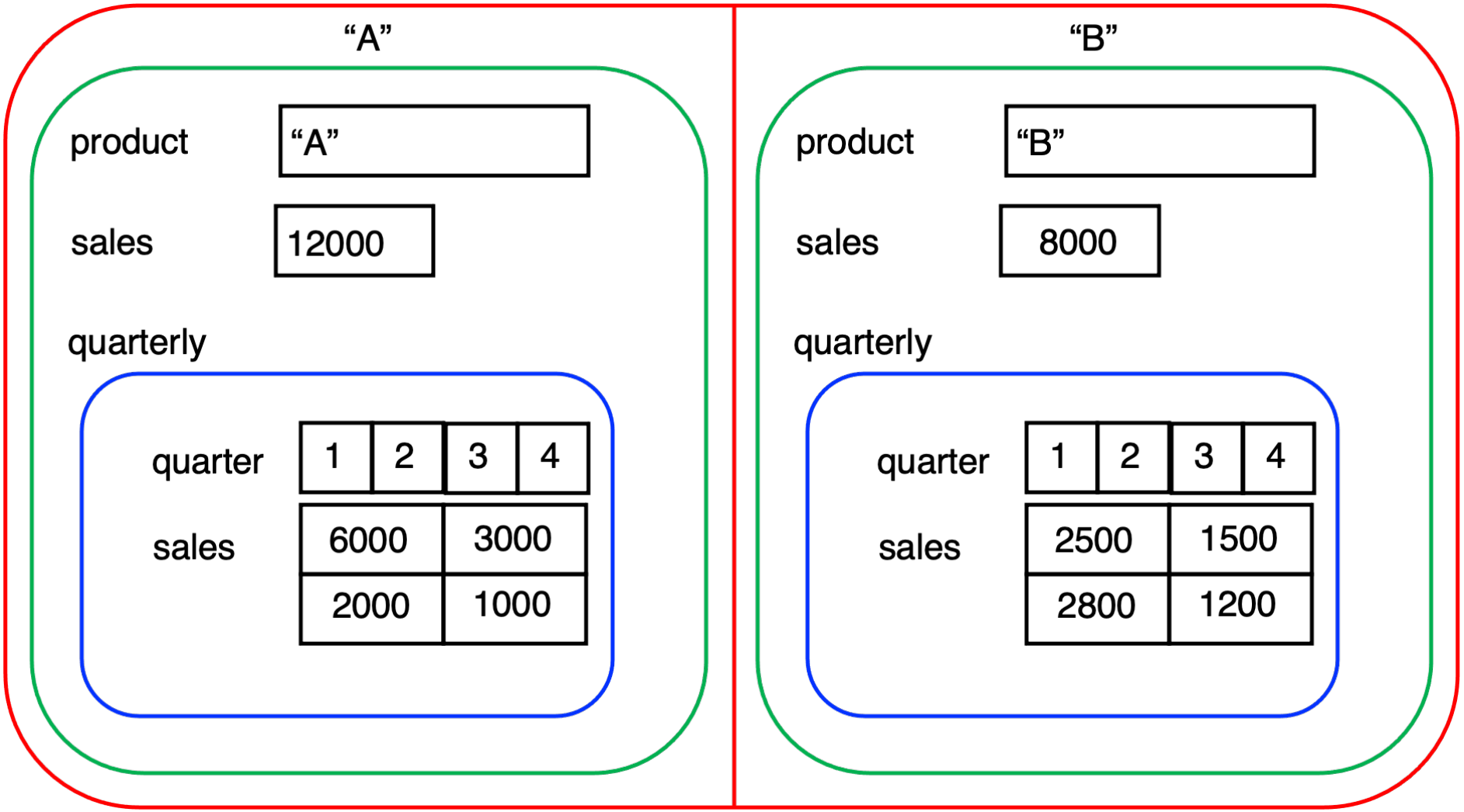


Figure 1: Visualisation of the products data structure

Some observations on the list

- At its core, the list is simply a vector of two named elements, and this is highlighted with the red lines.
- We can check this with R code to show (1) the length of the vector and (2) the name of each element.

```
# Show the vector length (2 elements)
length(products)
```

```
## [1] 2
```

```
# Show the names of each element
names(products)
```

```
## [1] "A" "B"
```

Some observations on the list

- However, even though there are just two elements in the list, each element has a significant internal structure.
- Each element contains a list, highlighted in green. This list contains three elements:
  - the product name, a character atomic vector,
  - the sales, a numeric atomic vector
  - an element named quarterly, which is another list (coloured blue).
- This third list contains two atomic vector elements:
  - the quarter number (1, 2, 3 and 4), and
  - the corresponding sales amount for each quarter, for the product.
- Note that the sum of the `sales` vector in this list equals the amount in the `sales` vector in the previous list.

Subsetting lists

- Subsetting lists is more challenging than subsetting atomic vectors.
- There are three methods that can be used, and to illustrate the core idea we define a list (l1) that contains three named elements.

```
# Create a simple list 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"
```

Subsetting with [

- The single square bracket '[', when applied to a list, will *always return a list*.
- The same indexing method used for atomic vectors can also be used for filtering lists, namely: positive integers, negative integers, logical vectors, and the element name. Here are examples of filtering a list using each of these methods.

```
# extract the first element of the list
str(l1[1])
```

```
## List of 1
##  $ a: chr "Hello"
```

```
str(l1["a"])
```

```
## List of 1
##  $ a: chr "Hello"
```

Subsetting with [

Some additional examples:

```
# extract the first two list elements
str(l1[1:2])
```

```
## List of 2
##  $ a: chr "Hello"
##  $ b: int [1:5] 1 2 3 4 5
```

```
str(l1[c(T,F)])
```

```
## List of 2
##  $ a: chr "Hello"
##  $ c:List of 2
##   ..$ d: logi [1:3] TRUE TRUE FALSE
##   ..$ e: chr "Hello World"
```

Extracting list contents with [[

- The single bracket [return a list, but in many cases this is not sufficent for analysis
- We will need to access the data within the list (which can be an atomic vector, and also a list).
- For example, finding the value of element a or element b.
- To do this, we must use the '[[' operator, which extracts the *contents of a list* at a given location (i.e. element 1, 2, .., N), where N is the list length.

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

```
## [1] "Hello"
```

```
l1[["a"]]
```

```
## [1] "Hello"
```

Extracting list contents with [[

```
# extract the contents of the second  list element
l1[["b"]]
```

```
## [1] 1 2 3 4 5
```

```
# 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"
```

```
l1[["c"]][["d"]]
```

```
## [1]  TRUE  TRUE FALSE
```

```
l1[[3]][[1]]
```

```
## [1]  TRUE  TRUE FALSE
```

Extracting list contents with $

- There is a convenient alternative to the [[operator, and this is the tag operator $ which can be used once a list element is named.
- For example, for our list l1 the terms l1[[1]], l1[["a"] and l1$a are the same, and in the general case l[["y]] is equivalent to l$y

```
l1$b
```

```
## [1] 1 2 3 4 5
```

```
# extract the contents of the third list element (a list!)
str(l1$c)
```

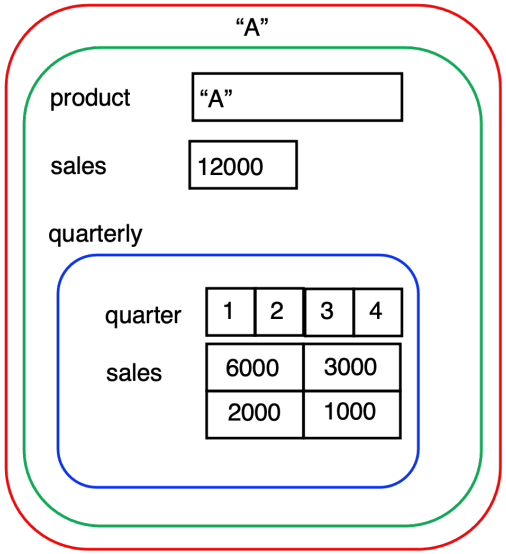
```
## List of 2
##  $ d: logi [1:3] TRUE TRUE FALSE
##  $ e: chr "Hello World"
```

```
l1$c$d
```

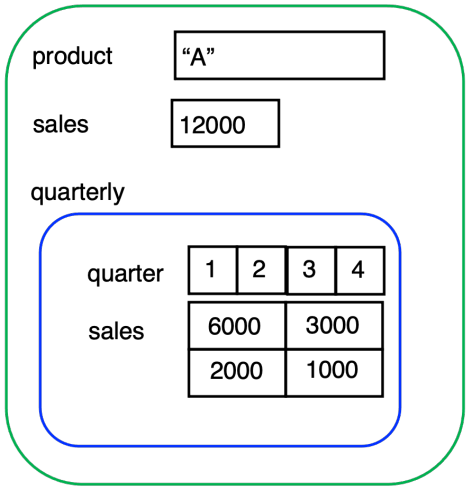
```
## [1]  TRUE  TRUE FALSE
```

Visualising subsets of `products`

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

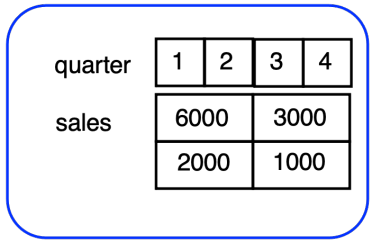


(2) products[[1]] *or* products[["A"]] *or* products$A

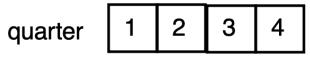


(4) products[[1]][[2]] *or* products[["A"]][["sales"]] *or* products$A$sales

(5) products[[1]][[3]] *or* products[["A"]][["quarterly"]] *or* products$A$quarterly



(6) products[[1]][[3]][[1]] *or* products[["A"]][["quarterly"]][["quarter"]] *or* products$A$quarterly$quarter



(7) products[[1]][[3]][[2]] or products[["A"]][["quarterly"]][["sales"]] *or* products$A$quarterly$sales

(8) products[[1]][[3]][[2]][1:2] *or*

Code for (1) - get the first list element

```
# 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
```

Code for (2) - get the contents of the first list element

```
# 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
```

Code for (3) - get the product name from the 1st list element

```
# Example (3) - get the product name for the first product
ex3.1 <- products[[1]][[1]]
ex3.2 <- products[["A"]][["product"]]
ex3.3 <- products$A$product
str(ex3.1)
```

```
##  chr "A"
```

Code for (4) - get annual sales of Product A

```
# Example (4) - get the annual sales for the first product
ex4.1 <- products[[1]][[2]]
ex4.2 <- products[["A"]][["sales"]]
ex4.3 <- products$A$sales
str(ex4.1)
```

```
##  num 12000
```

Code for (5) - get the list for quarterly sales data

```
# Example (5) - get as a list, the detailed quarterly sales
ex5.1 <- products[[1]][[3]]
ex5.2 <- products[["A"]][["quarterly"]]
ex5.3 <- products$A$quarterly
str(ex5.1)
```

```
## List of 2
##  $ quarter: int [1:4] 1 2 3 4
##  $ sales  : num [1:4] 6000 3000 2000 1000
```

Code for (6) - get the quarters vector

```
# Example (6) - get the quarters
ex6.1 <- products[[1]][[3]][[1]]
ex6.1 <- products[["A"]][["quarterly"]][["quarter"]]
ex6.1 <- products$A$quarterly$quarter
str(ex6.1)
```

```
##  int [1:4] 1 2 3 4
```

Code for (7) - get the quarterly sales vector

```
# Example (7) - get the quarterly sales
ex7.1 <- products[[1]][[3]][[2]]
ex7.1 <- products[["A"]][["quarterly"]][["sales"]]
ex7.1 <- products$A$quarterly$sales
str(ex7.1)
```

```
##  num [1:4] 6000 3000 2000 1000
```

Code for (8) - subset the quarterly sales for product A

```
# Example (8) - get the quarterly sales for the first two quar
ex8.1 <- products[[1]][[3]][[2]][1:2]
ex8.2 <-products[["A"]][["quarterly"]][["sales"]][1:2]
ex8.3 <-products$A$quarterly$sales[1:2]
str(ex8.1)
```

```
##  num [1:2] 6000 3000
```

Updating and adding new elements

```
# Increase the sales of product A by 10,000
products$A$sales <- products$A$sales + 10000
# Add a new field to product A
products$A$type <- "Food"
str(products$A)
```

```
## List of 4
##  $ product  : chr "A"
##  $ sales    : num 22000
##  $ quarterly:List of 2
##   ..$ quarter: int [1:4] 1 2 3 4
##   ..$ sales  : num [1:4] 6000 3000 2000 1000
##  $ type     : chr "Food"
```

Some observations on list subsetting

- Clearly, for list manipulation, the tag operator is the most programmer-friendly, so it is recommended to use this, and also try and ensure that the list elements are named
- Indexing using `[[` by positive integer is very useful for looping structures, we will see an example of this shortly
- Functions such as those in the package `purrr` provide efficient and flexible ways to iterate through lists.

Iteration using Loops

- 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, and we will focus on one of these, 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 (a list or an atomic vector), is to use the function `seq_along()` which returns the indices of a vector.

Example loop structure (Atomic Vector)

```
set.seed(100)
(v <- sample(1:6,10,replace = T))
```

```
##  [1] 2 6 3 1 2 6 4 6 6 4
```

```
seq_along(v)
```

```
##  [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
```

Example loop structure using [[(List)

Our goal is to find the average sales for the two products, and for this we can use a list.

```
sum_sales <- 0

for(i in seq_along(products)){
  cat("Sales of product ",i," = ",products[[i]]$sales,"\n")
  sum_sales <- sum_sales+products[[i]]$sales
}
```

```
## Sales of product  1  =  22000
## Sales of product  2  =  8000
```

```
(avr_sales <- sum_sales / length(products))
```

```
## [1] 15000
```

The if statement

- `if(cond) expr` which evaluates `expr` if the condition `cond` is true
- `if(cond) true.expr else false.expr`, which evaluates `true.expr` if the condition is true, and otherwise evaluates `false.expr`

```
# create a test vector
v <- 1:10
lv <- vector(mode="logical",length(v))
for(i in seq_along(v)){
  if(v[i] > mean(v))
    lv[i] <- TRUE
  else
    lv[i] <- FALSE
}
v[lv]
```

```
## [1]  6  7  8  9 10
```

Mini-Case Star Wars Movies

- The CRAN package `repurrrsive` is used
- It has lists on films (`sw_films`), people (`sw_people`), planets (`sw_planets`), species (`sw_species`), starships (`sw_starships`) and species (`sw_species`).
- Here we focus on the `sw_films` list, which contains seven elements, and each element is a list that contains fourteen elements.

```
library(repurrrsive)
length(sw_films)
```

```
## [1] 7
```

The first movie

```
str(sw_films[1])
```

```
## List of 1
##  $ :List of 14
##   ..$ title         : chr "A New Hope"
##   ..$ episode_id    : int 4
##   ..$ opening_crawl: chr "It is a period of civil war.\r\nR
##   ..$ director      : chr "George Lucas"
##   ..$ producer      : chr "Gary Kurtz, Rick McCallum"
##   ..$ release_date  : chr "1977-05-25"
##   ..$ characters    : chr [1:18] "http://swapi.co/api/people
##   ..$ planets       : chr [1:3] "http://swapi.co/api/planets
##   ..$ starships     : chr [1:8] "http://swapi.co/api/starshi
##   ..$ vehicles      : chr [1:4] "http://swapi.co/api/vehicle
##   ..$ species       : chr [1:5] "http://swapi.co/api/species
##   ..$ created       : chr "2014-12-10T14:23:31.880000Z"
##   ..$ edited        : chr "2015-04-11T09:46:52.774897Z"
```

Movie Directors for 1st and last movie

```
# Get the first film name and movie director
sw_films[[1]][[1]]
```

```
## [1] "A New Hope"
```

```
sw_films[[1]][[4]]
```

```
## [1] "George Lucas"
```

```
# Get the last film name and movie director
sw_films[[length(sw_films)]][[1]]
```

```
## [1] "The Force Awakens"
```

```
sw_films[[length(sw_films)]][[4]]
```

```
## [1] "J. J. Abrams"
```

Movie Directors for 1st and last movie

```
# Get the first film name and movie director
sw_films[[1]]$title
```

```
## [1] "A New Hope"
```

```
sw_films[[1]]$director
```

```
## [1] "George Lucas"
```

```
# Get the last film name and movie director
sw_films[[length(sw_films)]]$title
```

```
## [1] "The Force Awakens"
```

```
sw_films[[length(sw_films)]]$director
```

```
## [1] "J. J. Abrams"
```

Find all movies directed by George Lucas

- A for-loop structure (along with `seq_along()`)is used to iterate over the entire loop and mark those elements as either a match (`TRUE`) or not a match (`FALSE`). This information is stored in an atomic vector.
- Before entering the loop, we create a logical vector variable (`is_target`) of size seven (the same size as the list), and this will store information on whether a list item should be marked for further processing.
- For each list element we extract the directors name and check if it matches the target ("George Lucas"), and store this value in the corresponding element of `is_target`.
- The vector `is_target` can then be used to filter the original `sw_films` list and retain all the movies directed by George Lucas.

Code Solution

```
# Search for movies by George Lucas and store these in a new
target <- "George Lucas"
# Create a logical vector that will hold information for posi
is_target <- vector(mode="logical",length = length(sw_films))
# Iterate through the entire sw_films list (of 7)
for(i in seq_along(sw_films)){
  is_target[i] <- sw_films[[i]]$director == target
}
is_target
```

```
## [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

```
target_list <- sw_films[is_target]
length(target_list)
```

```
## [1] 4
```

Useful R Functions (1/2)

R Function	Description
`as.vector()`	Coerces its argument into a vector
`c()`	Used to create an atomic vector, with elements separated
`head()`	Lists the first six values of a data structure
`is.logical()`	A test to see if a variable is a logical type
`is.integer()`	A test to see if a variable is an integer type
`is.double()`	A test to see if a variable is a double type
`is.character()`	A test to see if a variable is a character type
`is.na()`	A function to test for the presence of NA values
`ifelse()`	An if-else vectorised function that operates on atomic ve
`length()`	Returns the length of an atomic vector or list
`mean()`	Calculates the mean of a vector
`names()`	Can be used to show the vector names, or set the vector
`str()`	Compactly displays the internal structure of a variable
`set.seed()`	Provides a way to initialize a pseudorandom number gene

Useful R Functions (2/2)

R Function	Description
`sample()`	Generates a random sample of values, with or without replacem
`summary()`	A function that provides a useful summary of a variable
`tail()`	Lists the final six values of a data structure
`table()`	Builds a table of frequency data from an input atomic vector
`typeof()`	Displays the atomic vector type
`unlist()`	Converts a list to an atomic vector

Exercise 1

Predict what the types will be for the following variables, and then verify your results in R.

```
v1 <- c(1L, FALSE)
v2 <- c(1L, 2.0, FALSE)
v3 <- c(2.0, FALSE, "FALSE")
v4 <- c(1:20, seq(1,10,by=.5))
v5 <- unlist(list(1:10,list(11:20,"Hello")))
```

Exercise 2

Create the following atomic vector, which is a combination of the character string *St* and a sequence of numbers from 1 to 7. Explore how the R function `paste0()` can be used to generate the solution. Type `?paste0` to check out how this function can generate character strings.

```
# The output generated following the call to paste0()
slist
```

```
## [1] "St-1" "St-2" "St-3" "St-4" "St-5" "St-6" "St-7"
```

Exercise 3

Generate a random sample of 20 temperatures (assume integer values in the range -5 to 30) using the `sample()` function (`set.seed(99)`). Assume that temperatures less than 4 are cold, temperatures greater that 25 are hot, and all others are medium, use the `ifelse()` function to generate the following vector. Note that an `ifelse()` call can be nested within another `ifelse()` call.

```
# The temperature data set
temp[1:6]
```

```
## [1] 27 16 29 28 26  7
```

```
# The descriptions for each temperature generated by ifelse()
des[1:6]
```

```
## [1] "Hot"    "Medium" "Hot"    "Hot"    "Hot"    "Medium"
```

Lecture Summary

- The difference between an atomic vector and a list, and be able to create atomic vectors and lists using the `c()` and `list()` functions.
- The four main types of atomic vector, and how different vector elements can be named.
- The rules of coercion for atomic vectors, and the importance of the function `is.na()`
- The idea of vectorisation, and how arithmetic and logical operators can be applied to vectors.
- Key R functions that allow you to work with vectors.
- How to solve all three challenges