1. Base R - Vectors

CT5102 - J. Duggan (University of Galway)

R

- R's mission is to enable the best and most thorough exploration of data possible (Chambers 2008).
- It is a dialect of the S language, developed at Bell Laboratories
- ACM noted that S "will forever alter the way people analyze, visualize, and manipulate data"
- See https://github.com/JimDuggan/CT5102

```r
x <- 2 + 3
x
```

```
## [1] 5
```

Vectors

> *The vector type is really the heart of R. It's hard to imagine R code, or even an interactive R session, that doesn't involve vectors.*
> *— Norman Matloff "The Art of R Programming"*

Overview

- A vector is a one-dimensional data structure in R
- There are two types of vectors: **atomic vectors**, where all the data must be of the same type, and **lists**, which are more flexible and each element's type can vary.
- Learning outcomes:
  - 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

Atomic Vectors

- The *atomic vector* is a powerful one-dimensional data structure that allows you to store one or more values.
- Note that unlike other languages, there is no special variable in R for storing a single value.
- A single variable (often called a *scalar*) in R is simply an atomic vector of size 1.
- For atomic vectors all of the elements must be of the same type.
- In order to create an atomic vector, the function `c()` is used (short for combine).
- When assigning a value to a variable, R's *right-to-left* assignment operator <- should be used
- There are 4 main types: **double**, **integer**, **logical** and **character**

```
v1 <- c(2.9, 3.1, 4.8)
v1
```

```
## [1] 2.9 3.1 4.8
```

Some useful functions

- `typeof()`, which shows the variable type, which will be one of the four categories.
- `str()`, which compactly displays the internal structure of a variable, and also shows the type. This is a valuable function that you will make extensive use of in R, particularly when you are exploring data returned from functions.
- `is.logical()`, is.double()`,`is.integer()`, and`is.character()`which tests the variable's type, and returns`TRUE‘ if the type matches the function name.

Logical type

- **Logical**, where values can be either TRUE or FALSE
- The abbreviations T and F can also be used.
- For example, here we declare a logical vector with five elements.

```
x_logi <- c(TRUE, T, FALSE, TRUE, F)
x_logi
```

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

```
typeof(x_logi)
```

```
## [1] "logical"
```

```
str(x_logi)
```

```
##  logi [1:5] TRUE TRUE FALSE TRUE FALSE
```

Integer type

- **Integer**, which represent whole numbers (negative and positive), and must be declared by appending the letter `L` to the number.
- `L` is that it is an abbreviation for the word *long*, which is used to define integers in R.

```
# Create an integer vector
x_int <- c(2L, 4L, 6L, 8L, 10L)
x_int
```

```
## [1]  2  4  6  8 10
```

```
typeof(x_int)
```

```
## [1] "integer"
```

```
str(x_int)
```

```
##  int [1:5] 2 4 6 8 10
```

Double type

- **Double** represents floating point numbers.
- Note that integer and double vectors are also known as numeric vectors

```
# Create a double vector
x_dbl<- c(1.2, 3.4, 7.2, 11.1, 12.7)

x_dbl
```

```
## [1]  1.2  3.4  7.2 11.1 12.7
```

```
typeof(x_dbl)
```

```
## [1] "double"
```

```
str(x_dbl)
```

```
##  num [1:5] 1.2 3.4 7.2 11.1 12.7
```

Character type

- **Character**, which represents values that are in string format.

```
## [1] "One"   "Two"   "Three" "Four"  "Five"
```

```
## [1] "character"
```

```
##  chr [1:5] "One" "Two" "Three" "Four" "Five"
```

Appending atomic vectors

```r
# Create vector 1
v1 <- c(1,2,3)
# Create vector 2
v2 <- c(4,5,6)

# Append for vector 3
v3 <- c(v1, v2)
v3
```

```
## [1] 1 2 3 4 5 6
```

```r
# Append for vector 4
v4 <- c(v2, v1)
v4
```

```
## [1] 4 5 6 1 2 3
```

Creating larger vectors

- While the `c()` function creates atomic vectors, there may be cases where a larger atomic vector needs to be constructed, and there are a number of additional ways that can be used for this.

- The colon operator `:` which generates a regular sequence as an atomic vector, from a starting value to the end of a sequence. The function `length()` can also be used to confirm the number of elements in the atomic vector.

```
x <- 1:10


x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
typeof(x)
```

```
## [1] "integer"
```

The seq() function

- The sequence function `seq()`, generates a regular sequence from a starting value (`from`) to a final value (`to`).
- It also allows for an increment between each value (`by`), and for a fixed length for the vector to be specified (`length.out`).
- Note that when calling a function in R such as the `seq` function , the argument name (i.e. `from`)can be used when passing a value, this is useful, as it means we don't need to know that exact positioning of the argument in the parameter list.

seq() Examples

```
(x0 <- seq(1,10))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
(x1 <- seq(from=1, to=10))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
(x2 <- seq(from=1, to=5,by=.5))
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
(x3 <- seq(from=1, to=5,by=2))
```

```
## [1] 1 3 5
```

```
(x4 <- seq(length.out=10))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

The rep() function

- The replication function `rep()` replicates values contained in the input vector, a given number of times.

```r
# use rep to initialise a vector to zeros
z1 <- rep(0,10)
z1
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```r
# use rep to initialise with a pattern of values
z2 <- rep(c("A","B"),5)
z2
```

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

Creating Vectors of fixed size (in advance)

- The `vector()` function creates a vector of a fixed length, and initialises each vector element
- This can be useful for creating larger vectors in advance of carrying out processing operations on each individual element.

```
(y1 <- vector("logical",   length = 3))
```

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

```
(y2 <- vector("integer",   length = 3))
```

```
## [1] 0 0 0
```

```
(y3 <- vector("double",   length = 3))
```

```
## [1] 0 0 0
```

```
(y4 <- vector("character", length = 3))
```

Coercion of atomic vectors

- All elements of an atomic vector **MUST** be of the same type
- When different type are combined, they will be coerced into the most flexible types

	logical	**integer**	**numeric**	**character**
logical	logical	integer	numeric	character
integer	integer	integer	numeric	character
numeric	numeric	numeric	numeric	character
character	character	character	character	character

Coercion Examples 1

```r
(ex1 <- c(T,F,T,7L))
```

```
## [1] 1 0 1 7
```

```r
typeof(ex1)
```

```
## [1] "integer"
```

```r
(ex2 <- c(T,F,T,7.3))
```

```
## [1] 1.0 0.0 1.0 7.3
```

```r
typeof(ex2)
```

```
## [1] "double"
```

Coercion Examples 2

```
(ex3 <- c(1L,2L, 3L, 4.1))
```

```
## [1] 1.0 2.0 3.0 4.1
```

```
typeof(ex3)
```

```
## [1] "double"
```

```
(ex4 <- c(TRUE,1L,2.0, "Hello"))
```

```
## [1] "TRUE"  "1"     "2"     "Hello"
```

```
typeof(ex4)
```

```
## [1] "character"
```

Naming Atomic Vectors

- A highly useful feature of R is that vector elements (both atomic vectors and lists) can be named
- Names are an *attribute* of vectors.
- There are a number of ways to name vector elements, and the simplest is to declare the name of each element as part of the c() function, using the = symbol.
- Note that the names do not have any impact on manipulating the data.

```
# Create a double vector with named elements
(x_dbl<- c(a=1.2, b=3.4, c=7.2, d=11.1, e=12.7))
```

```
##    a    b    c    d    e
##  1.2  3.4  7.2 11.1 12.7
```

Extracting the names

- A character vector of the element names can be easily extracted using a special R function called `names()`
- First, we can see how the element names are extracted.

```r
# Extract the names of the x_dbl vector
x_dbl_names <- names(x_dbl)

typeof(x_dbl_names)
```

```
## [1] "character"
```

```r
x_dbl_names
```

```
## [1] "a" "b" "c" "d" "e"
```

Setting names using the `names()` function

- `names()` can act as an *accessor function*, that will return the names of the vector elements
- It can also be used to set the names of a vector.

```
# Show our previously defined vector x_logi
x_logi
```

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

```
# Allocal names to each vector element
names(x_logi) <- c("f","g","h","i","j")
x_logi
```

```
##     f     g     h     i     j
##  TRUE  TRUE FALSE  TRUE FALSE
```

Missing Values - Introducing *NA*

- When analysing data, it is common that there will be missing values.
- For example, a sensor such as a thermometer might break down on any given day, and so an hourly temperature recording could be missed.
- In R, `NA` - short for *not applicable* - is a logical constant of length one which contains a missing value indicator.

```r
(v <- 1:5)
```

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

```r
v[5] <- NA

v
```

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

Are there NAs in a vector?

- How can we check whether NA is present in a vector?
- The function is.na() is used
- It returns returns a logical vector showing whether an NA value is present at a given location

```
v
```

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

```
(is.na(v))
```

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

```
# sum the vector, why is the answer NA?
sum(v)
```

```
## [1] NA
```

NAs get in the way!

- The presence of an `NA` value causes difficulty for what should be straightforward calculation.
- This obstacle can be overcome by adding an additional argument to the `max()` function which is `na.rm=TRUE`
- R then ignores all the `NA` values

```
v
```

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

```
sum(v)
```

```
## [1] NA
```

```
(sum(v, na.rm = TRUE))
```

```
## [1] 10
```

Vectorisation

- Vectorisation is a powerful feature of R
- A function can operate on all the elements of an atomic vector, and return all the results in new atomic vector, of the same size.
- We have:
  - input atomic vector of size N with elements $x_1, x_2, x_3, ..., x_N$
  - a vectorised function $f(x)$ that will (1) create an output atomic vector of size N of the appropriate type (it does not have to be the same type as the input), and (2) loop through each element, apply the code, and store the result in the output vector $f(x_1), f(x_2), f(x_3), ..., f(x_N)$

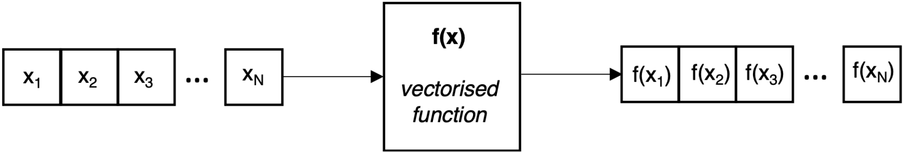

Figure 1: Vectorisation in R

A vectorised example `sqrt()`

- Many of R's functions are vectorised, which means that they perform the same operation on each data element, and store the result in a new vector.
- The function `sqrt()` is a nice example of a vectorised function, as it takes in a vector of numbers, and calculates the square root of each individual number. This is shown in the following code.

```r
# Set the random number seed to 100
set.seed(100)
# Create a sample of 5 numbers from 1-10.
(v <- sample(1:10,5))
```

```
## [1] 10  7  6  3  1
```

```r
(sv <- sqrt(v))
```

```
## [1] 3.162278 2.645751 2.449490 1.732051 1.000000
```

Arithmetic Operators

- Similar to other programming languages, R has arithmetic, relational and logical operators.
- R's main arithmetic operators are shown below.
- Note, in cases where two vectors are of unequal length, R has a *recycling* mechanism, where the shorter vector will be recycled in order to match the longer vector

R Arithmetic Operator	Description
+	Addition
−	Subtraction
∗	Multiplication
/	Division
%/%	Integer division
∗∗ or ˆ	Exponentiation
%%	Modulus

Arithemic Operators - 1

```
(v1 <- c(10, 20, 30))
```

```
## [1] 10 20 30
```

```
(v2 <- c(2, 4, 3))
```

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

```
v1 + v2
```

```
## [1] 12 24 33
```

```
v1 - v2
```

```
## [1]  8 16 27
```

```
v1 * v2
```

```
## [1] 20 80 90
```

```
v1 / v2
```

Arithemic Operators - 2

```
v1 / v2
```

```
## [1]  5  5 10
```

```
v1 ^ v2
```

```
## [1]    100 160000  27000
```

```
v1 %/% 7
```

```
## [1] 1 2 4
```

```
v1 %% 7
```

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

Relational Operators

- Relational operators allow for comparison between values, and *always return a logical vector*.
- The output from the use of a relational operator is a logical atomic vector that is the same size as the input vector, and every value contains the result of the logical operation applied to the corresponding element
- There are six categoires of relation operators, as shown in this table.

R Relational Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Relational Operator Examples - 1

```r
# Setup a test vector
(v5 <- c(5,1,4,2,6,8))
```

```
## [1] 5 1 4 2 6 8
```

```r
v5 < 4
```

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

```r
v5 <= 4
```

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

```r
v5 > 4
```

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

```r
v5 >= 4
```

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

Relational Operator Examples - 2

```
v5
```

```
## [1] 5 1 4 2 6 8
```

```
v5 == 4
```

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

```
v5 != 4
```

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

```
# Recycling also works
v5 == c(5,8)
```

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

Summarising Relational Operator Results

- Relational operators can take advantage of R's coercion rules, can be conveniently used with the `sum()` function
- Logical values are coerced to ones or zeros.
- For example, we can see how many vector elements are greater than the mean.

```
(v6 <- 1:7)
```

```
## [1] 1 2 3 4 5 6 7
```

```
l_test <- v6 > mean(v6)
l_test
```

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

```
sum(l_test)
```

```
## [1] 3
```

Logical Operators

- Logical operators perform logical operations, for example `AND` and `NOT`
- For atomic vectors, the following logical operators can be used.

R Logical Operator	Description
`!`	Logical NOT: Converts TRUE to FALSE, or FALSE to TRUE
`&`	Logical AND: TRUE if all relational expressions are TRUE, otherwise FALSE
`|`	Logical OR: TRUE if any relational expression is TRUE, otherwise FALSE

Logical Operator Example

```r
set.seed(200)
(v  <- sample(1:20, 5, replace = T))
```

```
## [1]  6 18 15  8 12
```

```r
v >= 10 & v <= 14
```

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

```r
v < 5 | v > 17
```

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

```r
!(v %% 2 == 0)
```

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

ifelse() vectorisation

The `ifelse()` function allows for successive elements of an atomic vector to be processed with the same conditional logic. The general form of the function is `ifelse(test_condition, true_value, false_value)`, where:

- `test_condition` is a logical vector, or an operation that yields a logical vector, such as a logical operator
- `true_value` is the new vector value if the condition is true
- `false_value` is the new vector value if the condition is false

```
(v <- 1:5)
```

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

```
(test <- ifelse(v %% 2 == 0,"Even","Odd"))
```

```
## [1] "Odd"  "Even" "Odd"  "Even" "Odd"
```

Mini-Case: Two-dice rolls with Atomic Vectors

Our goal is to simulate this, and check the proportions with N=1000

Dice Rolls	Probability	Sum	Proportion
(1,1)	1/36	2	0.02777778
(1,2)(2,1)	2/36	3	0.05555556
(1,3)(3,1)(2,2)	3/36	4	0.08333333
(1,4)(4,1)(2,3)(3,2)	4/36	5	0.1111111
(1,5)(5,1)(2,4)(4,2)(3,3)	5/36	6	0.1388889
(1,6)(6,1)(2,5)(5,2)(4,3) (3,4)	6/36	7	0.1666667
(2,6)(6,2)(3,5)(5,3)(4,4)	5/36	8	0.1388889
(3,6)(6,3)(4,5)(5,4)	4/36	9	0.1111111
(4,6)(6,4)(5,5)	3/36	10	0.08333333
(3,6)(6,3)(4,5)(5,4)	2/36	11	0.05555556
(3,6)(6,3)(4,5)(5,4)	1/36	12	0.02777778

The function `table()` is useful for summarising integer data

```r
set.seed(200)
N <- 500
v <- sample(1:5,N,replace = T)
(freq <- table(v))
```

```
## v
##   1   2   3   4   5
##  94 101 117  93  95
```

```r
(prop <- table(v)/N)
```

```
## v
##     1     2     3     4     5
## 0.188 0.202 0.234 0.186 0.190
```

```r
(check <- sum(prop))
```

Lists

- A list is a vector that can contain different types, including a list.
- It is a flexible data structure, for example, `lm()`, returns a list containing all information relating to the results of a linear regression task.
- A list can be defined using the `list()` function, which is similar to the `c()` function used to create atomic vectors.

```
l1 <- list(1:2,c(TRUE, FALSE),list(3:4,5:6))

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
```

Visualising lists

While atomic vectors can be visualised as a rectangular set of elements, we use a rounded rectangle to visualise a list.

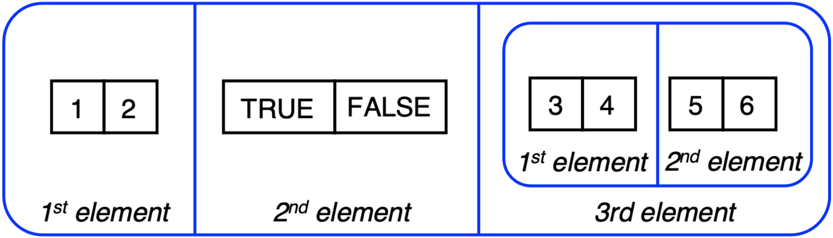

Figure 2: Visualising a list of 3 elements

Creating larger lists

- As we have seen with atomic vectors, it can be valuable to create a larger structure that could then be processed using a loop.
- With lists, the function `vector()`, which we have seen used for creating atomic vectors, can also be used to create a list of a given size.
- Note that the initial values are `NULL`

```r
l2 <- vector(mode="list",length=3)


str(l2)
```

```
## List of 3
##  $ : NULL
##  $ : NULL
##  $ : NULL
```

```r
length(l2)
```

```
## [1] 3
```

Naming List Elements

- Similar to atomic vectors, each element of a list can be named. Benefits include:
  - increased clarity when using the `str()` function
  - increased flexibility when accessing a list element

```
l2 <- list(el1=1:2,el2=c(TRUE, FALSE),el3=list(el3_1=3:4,el3_
str(l2)
```

```
## List of 3
##  $ el1: int [1:2] 1 2
##  $ el2: logi [1:2] TRUE FALSE
##  $ el3:List of 2
##   ..$ el3_1: int [1:2] 3 4
##   ..$ el3_2: int [1:2] 5 6
```

```
names(l2)
```

```
## [1] "el1" "el2" "el3"
```

Converting a list to an atomic vector

A list can be easily converted to an atomic vector using the `unlist()` function. This "flattens" the entire list structure, and then performs coercion to ensure that all the elements are of the same type. If we take the list `l2` just defined, we can explore the output.

```
## 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
```

```
## [1] 1 2 1 0 3 4 5 6
```

```
##  int [1:8] 1 2 1 0 3 4 5 6
```

More complex list example - linear regression

```r
mod <- lm(mpg~disp,data=mtcars)
coef(mod)
```

```
## (Intercept)        disp
## 29.59985476 -0.04121512
```

```r
str(mod)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 29.5999 -0.0412
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "disp"
##  $ residuals    : Named num [1:32] -2.01 -2.01 -2.35 2.43 3
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 W
##  $ effects      : Named num [1:32] -113.65 -28.44 -1.79 2.6
##   ..- attr(*, "names")= chr [1:32] "(Intercept)" "disp" ""
##  $ rank         : int 2
##  $ fitted.values: Named num [1:32] 23 23 25.1 19 14.8 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 W
```

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.

```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.

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

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

```r
# 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