CT5102: Programming for Data Analytics

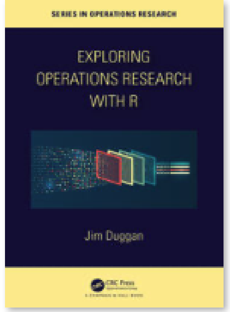
1. Vectors

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

https://github.com/JimDuggan/explore_or

Course Textbook (Accessible via University library)

Home > Computer Science > Software Engineering & Systems Development > Programming & Programming Languages > Exploring Operations Research with R



✓ FULL ACCESS

Accessibility Information

Book

Exploring Operations Research with R

By Jim Duggan

Edition	1st Edition
First Published	2024
eBook Published	21 May 2024
Pub. Location	New York
Imprint	Chapman and Hall/CRC
DOI	https://doi-org.nuigalway.idm.oclc.org/10.1201/9781003293910
Pages	395
eBook ISBN	9781003293910
Subjects	Computer Science, Engineering & Technology, Mathematics & Statistics

Share

Citation

You have download and read online access to this content.

DOWNLOAD

READ ONLINE

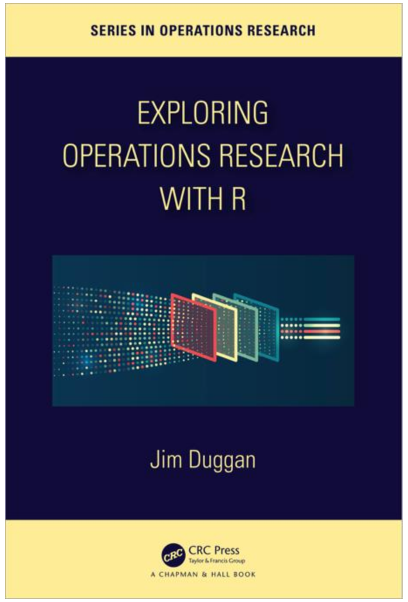
To purchase a print version of this book for personal use or request an inspection copy >>

GO TO ROUTLEDGE.COM

METRICS

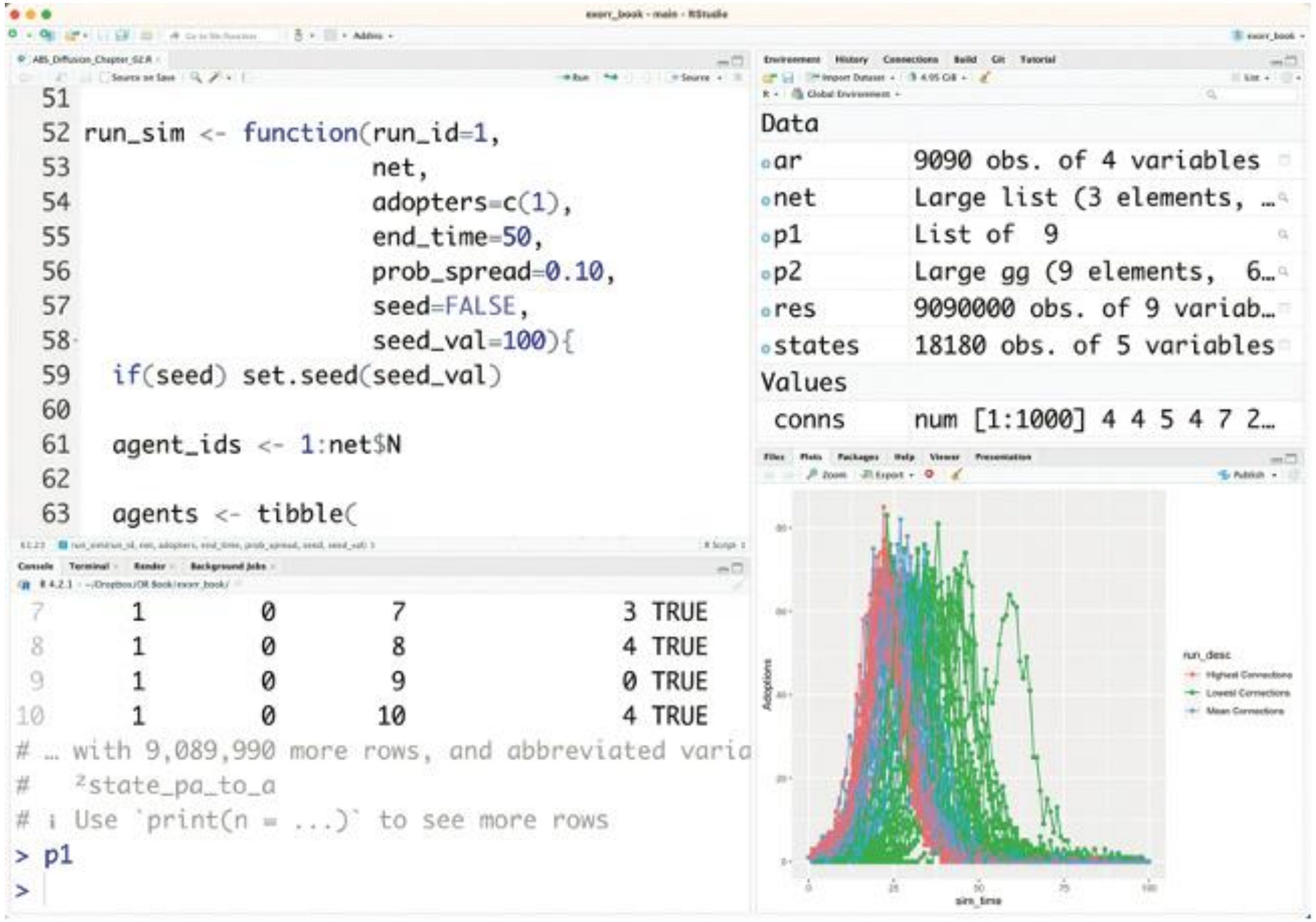
1

Citations



Course Overview

- Structure
  - Base R (Part I)
  - Tidyverse (Part II)
  - Applications (Part III)
- Tools
  - RStudio IDE (Projects)
  - Online (posit cloud)
  - On your device
- Marks breakdown
  - Years Work (Labs) – 40%
  - Written Exam – 60%

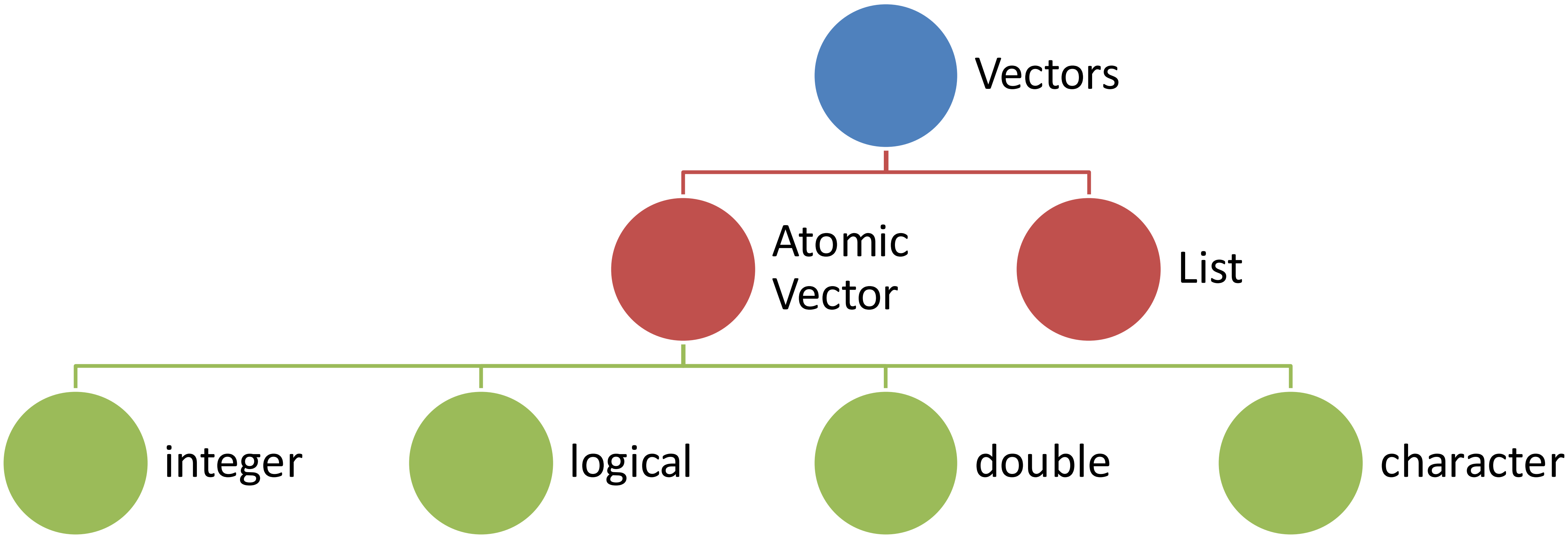


The vector type is really the heart of R. It's hard to imagine R code, or even an interactive R session, that doesn't involve vectors.

— Norman Matloff (Matloff, 2011)

(1.1) Atomic Vectors

- There are a number of data structures in R, and the first one we explore is the atomic vector.
- This is a one-dimensional data structure that allows you to store one or more values.
- Note that unlike other programming languages, there is no special variable in R for storing a single value. A single variable (often called a scalar in other languages) in R is simply an atomic vector of size 1.
- An important constraint of atomic vectors is that all of the elements must be of the same type.


Vectors
Atomic Vector
List
integer
logical
double
character

Atomic Vectors: Some useful functions

Function	Description
<--	Right-to-left assignment operator
c()	Creates an atomic vector
typeof()	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.
is.logical(), is.double(), is.integer(), and is.character()	which tests the variable's type, and returns the logical type TRUE if the type aligns with the function name

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

```r
# Create and display a logical vector
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

is.logical(x_logi)
#> [1] TRUE
```

integer, which represents whole numbers (negative and positive), and must be declared by appending the letter `L` to the number. The significance of `L` is that it is an abbreviation for the word *long*, which is a type of integer.

```
# Create and display 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

is.integer(x_int)
#> [1] TRUE
```

double, which represents floating point numbers. Note that integer and double vectors are also known as numeric vectors (Wickham, 2019).

```
# Create and display 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

is.double(x_dbl)
#> [1] TRUE
```

character, which represents values that are in string format.

```r
# Create and display a character vector
x_chr<- c("One","Two","Three","Four","Five")
x_chr
#> [1] "One"   "Two"   "Three" "Four"  "Five"

typeof(x_chr)
#> [1] "character"

str(x_chr)
#>  chr [1:5] "One" "Two" "Three" "Four" "Five"

is.character(x_chr)
#> [1] TRUE
```

Creating larger atomic vectors

- The colon operator : which generates a regular sequence of integers 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"
length(x)
#> [1] 10
```

- The sequence function `seq()`, which generates a regular sequence from a starting value (`from`) to a final value (`to`) and also allows for an increment between each value (`by`), or 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 can be used when passing a value. This is convenient, as it means we don't need to know that exact positioning of the argument in the parameter list. This will be discussed in more detail in Chapter 4.

```r
x0 <- seq(1,10)
x0
#>  [1]  1  2  3  4  5  6  7  8  9 10

x1 <- seq(from=1, to=10)
x1
#>  [1]  1  2  3  4  5  6  7  8  9 10

x2 <- seq(from=1, to=5, by=.5)
x2
#> [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

x3 <- seq(from=1, to=10, by=2)
x3
```

- The `vector()` function creates a vector of a fixed length. This is advantageous for creating larger vectors in advance of carrying out processing operations on each individual element. This function also initializes each vector element to a default value.

```
y1 <- vector("logical",   length = 3)
y1
#> [1] FALSE FALSE FALSE

y2 <- vector("integer",   length = 3)
y2
#> [1] 0 0 0

y3 <- vector("double",   length = 3)
y3
#> [1] 0 0 0

y4 <- vector("character", length = 3)
y4
#> [1] "" "" ""
```

Atomic vector - coercion

- In an atomic vector, all elements must be of the same type. If not, R will coerce the elements to the most flexible type.

	logical	**integer**	**double**	**character**
logical	logical	integer	double	character
integer	integer	integer	double	character
double	double	double	double	character
character	character	character	character	character

```r
# Create a vector with logical and integer combined
ex1 <- c(T,F,T,7L)
ex1
#> [1] 1 0 1 7
typeof(ex1)
#> [1] "integer"

# Create a vector with logical and double combined
ex2 <- c(T,F,T,7.3)
ex2
#> [1] 1.0 0.0 1.0 7.3
typeof(ex2)
#> [1] "double"
```

Naming vector elements

- An excellent feature of R is that vector elements (both atomic vectors and lists) can be named
- You can declare the name of each element as part of the c() function, using the = symbol.
- A character vector of the element names can be easily extracted using a special R function called names()
- names() can also be used to set the names on a vector

```r
# 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)

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

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

typeof(x_dbl_names)
#> [1] "character"

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

Missing Values - NA

- When working with real-world data, it is common that there will be missing values.
- For example, a thermometer might break down on any given day, causing an hourly temperature measurement to be missed.
- In R, the symbol NA is a logical constant of length one which contains a missing value indicator.
- Any value of a vector could have the value NA,

```r
# define a vector v
v <- 1:10
v
#>  [1]  1  2  3  4  5  6  7  8  9 10

# Simulate a missing value by setting the final value to NA
v[10] <- NA
v
#>  [1]  1  2  3  4  5  6  7  8  9 NA

# Notice how summary() deals with the NA value
summary(v)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#>       1       3       5       5       7       9       1
```

Is NA in a vector? - is.na()

```
v
#>  [1]  1  2  3  4  5  6  7  8  9 NA
```

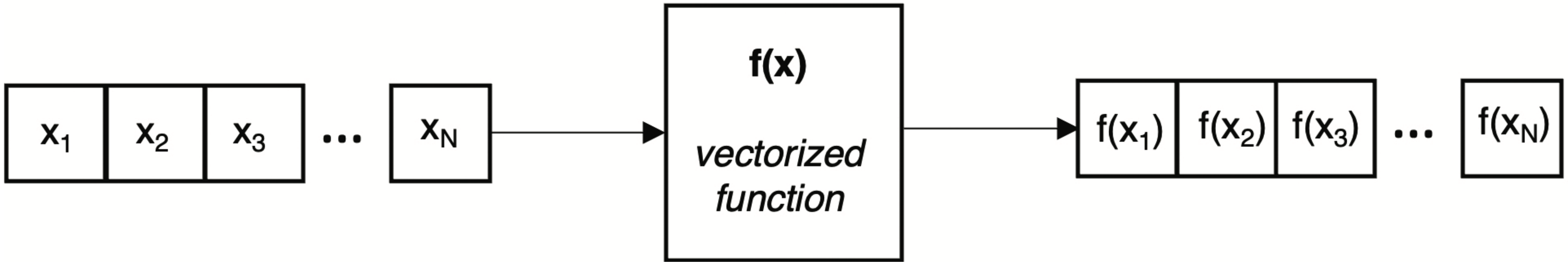
```
# Look for missing values in the vector v
is.na(v)
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
# Notice what happens when we try to get the maximum value of v
max(v)
#> [1] NA
```

```
max(v, na.rm = TRUE)
#> [1] 9
```

(1.2) Vectorization

- Vectorization is a powerful R feature that enables a function to operate on all the elements of an atomic vector, and return the results in new atomic vector, of the same size.
- Vectorization removes the requirement to write loop structures to iterate over the entire vector, and so it leads to a simplified data analysis process.
- Many R functions are vectorized



```r
# Set the random number seed to 100
set.seed(100)
# Create a sample of 5 numbers from 1-10.
# Numbers can only be selected once (no replacement)
v <- sample(1:10,5)
v
#> [1] 10  7  6  3  1
length(v)
#> [1] 5
typeof(v)
#> [1] "integer"

# Call the vectorized function sqrt (square root)
rv <- sqrt(v)
rv
#> [1] 3.162 2.646 2.449 1.732 1.000
```

R Arithmetic Operators – Support Vectorization

R Arithmetic Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%/%	Integer division
** or ^	Exponentiation
%%	Modulus

```r
# Define two sample vectors, v1 and v2
v1 <- c(10, 20, 30)
v1
#> [1] 10 20 30
v2 <- c(2, 4, 3)
v2
#> [1] 2 4 3

# Adding two vectors together
v1 + v2
#> [1] 12 24 33
```

Uneven length? - recycle

```r
# Define two unequal vectors
v3 <- c(12, 16, 20, 24)
v3
#> [1] 12 16 20 24
v4 <- c(2,4)
v4
#> [1] 2 4

# Recycling addition and subtraction
v3 + v4
#> [1] 14 20 22 28

v3 - v4
#> [1] 10 12 18 20
```

Relational Operators – Comparing Values

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

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

# Test for all six relational operators
v5 < 4
#> [1] FALSE  TRUE FALSE  TRUE FALSE FALSE

v5 <= 4
#> [1] FALSE  TRUE  TRUE  TRUE FALSE FALSE

v5 > 4
#> [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

Example… note use of sum() using coercion

```r
# Setup a test vector, in this case, a sequence
v6 <- 1:10
v6
#>  [1]  1  2  3  4  5  6  7  8  9 10

# create a logical test and see the results
l_test <- v6 > mean(v6)
l_test
#>  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

# Send the output to sum to see how many have matched
sum(l_test)
#> [1] 5
```

Logical Operators

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

```r
set.seed(200)
v  <- sample(1:20, 10, replace = T)
v
#>  [1]  6 18 15  8 12 18 12 20  8  4

# Use logical AND to see which values are in the range 10-14
v >= 10 & v <= 14
#>  [1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
```

ifelse() vectorization function

- The ifelse() function allows for successive elements of an atomic vector to be processed with the same test condition.
- The general form of the function is ifelse(test_condition, true_value, false_value)
  - 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

```r
# Create a vector of numbers from 1 to 10
v <- 1:10
v
#>  [1]  1  2  3  4  5  6  7  8  9 10

# Calculate the mean
m_v <- mean(v)
m_v
#> [1] 5.5

# Create a new vector des_v based on a condition, and using ifelse()
des_v <- ifelse(v > m_v, "GT", "LE")
des_v
#>  [1] "LE" "LE" "LE" "LE" "LE" "GT" "GT" "GT" "GT" "GT"
```

(1.3) Lists

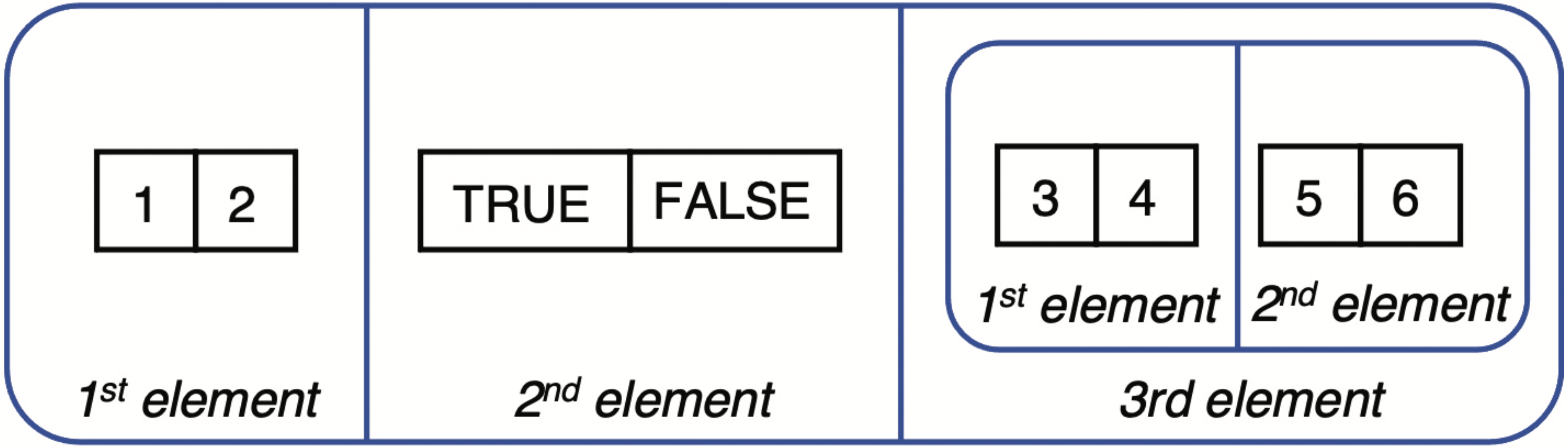
- A list is a vector that can contain different types, including a list.
- It is a flexible data structure and is often used to return data from a function.
- A list can be defined using the list() function, which is similar to the c() function used to create atomic vectors.

```r
# 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
```

```r
typeof(l1)
#> [1] "list"
# Summarize 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 (rounded rectangle)

```
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
```



Naming list elements (same as atomic vector)

```r
# Create a list
l1 <- list(el1=1:2,
           el2=c(TRUE, FALSE),
           el3=list(el3_el1=3:4,el3_el2=5:6))
# Summarize 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"
```

```r
# Create a list
l2 <- list(1:2,
           c(TRUE, FALSE),
           list(3:4,5:6))
# Name the list elements using names()
names(l2) <- c("el1","el2","el3")
str(l2)
#> List of 3
#>  $ el1: int [1:2] 1 2
#>  $ el2: logi [1:2] TRUE FALSE
#>  $ el3:List of 2
#>   ..$ : int [1:2] 3 4
#>   ..$ : int [1:2] 5 6
```

Converting a list to an atomic vector - *flattening*

```r
# Create a list
l3 <- list(1:4,c(TRUE, FALSE),list(2:3,6:7))
# Convert to an atomic vector
l3_av <- unlist(l3)
# Show the result and the type
l3_av
#>  [1] 1 2 3 4 1 0 2 3 6 7
typeof(l3_av)
#> [1] "integer"
```

(1.4) Atomic vector mini-case

- The aim of this example is to see how atomic vectors can be used to simulate the rolling of two dice, and to explore whether the expected frequency of outcomes is observed.
- The total sample space is 36 (6 × 6)
- The range of outcomes is (2,12)

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

Approach

- Create two vectors, one for each dice
- Use vectorized addition to create the solution vector
- Summarise the results, and also use R's table() function
- Use vectorized division to calculate probabilities
- Use the sample() function to "throw dice"

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

```r
# set the seed to 100, for replicability
set.seed(100)
# Create a variable for the number of throws
N <- 10000
# generate a sample for dice 1
dice1 <- sample(1:6, N, replace = T)
# generate a sample for dice 2
dice2 <- sample(1:6, N, replace = T)
```

```r
# Information on dice1
head(dice1)
#> [1] 2 6 3 1 2 6
summary(dice1)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    1.00    2.00    3.00    3.49    5.00    6.00
# Information on dice2
```

```r
# Create a new variable dice_sum, a vectorized sum of both dice.
dice_sum <- dice1 + dice2
head(dice_sum)
#> [1]  6 11  5  6  6  8

summary(dice_sum)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    2.00    5.00    7.00    7.01    9.00   12.00
```

```r
# Show the frequencies for the summed values
freq <- table(dice_sum)
freq
#> dice_sum
#>    2    3    4    5    6    7    8    9   10   11   12
#>  274  569  833 1070 1387 1687 1377 1165  807  534  297
```

```r
# Show the frequency proportions for the summed values,
# using the vectorized division operator
sim_probs <- freq/length(dice_sum)
sim_probs
#> dice_sum
#>      2      3      4      5      6      7      8      9     10
#> 0.0274 0.0569 0.0833 0.1070 0.1387 0.1687 0.1377 0.1165 0.0807
#>     11     12
#> 0.0534 0.0297
```

Sum	Proportion
2	0.02777778
3	0.05555556
4	0.08333333
5	0.1111111
6	0.1388889
7	0.1666667
8	0.1388889
9	0.1111111
10	0.08333333
11	0.05555556
12	0.02777778

Takeaways…

- The practical use of vectorization operations as part of the solution.
- The utility of R’s sample() function
- The use of the function set.seed(), as this allows for exact replication of results by others, and so supports sharing of insights
- When datasets are large, the use of summary() provides a summary description of data, and the functions head() and tail() provide a slice of data at the top, and at the end of the dataset

(1.5) Summary Functions

Function	Description
c()	Create an atomic vector.
head()	Lists the first six values of a data structure.
is.logical()	Checks whether a variable is of type logical.
is.integer()	Checks whether a variable is of type integer.
is.double()	Checks whether a variable is of type double.
is.character()	Checks whether a variable is of type character.
is.na()	A function to test for the presence of NA values.
ifelse()	Vectorized function that operates on atomic vectors.
list()	A function to construct a list.
length()	Returns the length of an atomic vector or list.

mean()	Calculates the mean for values in a vector.
names()	Display or set the vector names.
paste0()	Concatenates vectors after converting to a character.
str()	Displays the internal structure of a variable.
set.seed()	Initializes a pseudorandom number generator.
sample()	Generates a random sample of values.
summary()	Provides an informative summary of a variable.
tail()	Lists the final six values of a data structure.
table()	Builds a table of frequency data from a vector.
typeof()	Displays the atomic vector type.
unlist()	Converts a list to an atomic vector.

(1.6) Exercises

2. Create the following atomic vector, which is a combination of the character string *Student* and a sequence of numbers from 1 to 10.

   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] "Student-1"  "Student-2"  "Student-3"  "Student-4"
#>  [5] "Student-5"  "Student-6"  "Student-7"  "Student-8"
#>  [9] "Student-9"  "Student-10"
```

3. Use the R constants (character vectors) `LETTERS` and `letters` to generate the following list of four elements, where each list element is a sequence of six alphabetic characters. The names for each list element should be set based on the length of the list.

```
# The new list of four elements
str(l_list)
#> List of 4
#>  $ A: chr [1:6] "a" "b" "c" "d" ...
#>  $ B: chr [1:6] "g" "h" "i" "j" ...
#>  $ C: chr [1:6] "m" "n" "o" "p" ...
#>  $ D: chr [1:6] "s" "t" "u" "v" ...
```

4. 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 dataset
temp
#>  [1] 27 16 29 28 26  7 14 30 25 -2  3 12 18 24 16 14 26  8 -2  8

# The descriptions for each temperature generated by ifelse() call
des
#>  [1] "Hot"    "Medium" "Hot"    "Hot"    "Hot"    "Medium"
#>  [7] "Medium" "Hot"    "Medium" "Cold"   "Cold"   "Medium"
#> [13] "Medium" "Medium" "Medium" "Medium" "Hot"    "Medium"
#> [19] "Cold"   "Medium"
```

5. Use the expression `set.seed(100)` to ensure that you replicate the result as shown below. Configure a call to the function `sample()` that will generate a sample of 1000 for three categories of people: *Young*, *Adult*, and *Elderly*. Make use of the `prob` argument in `sample()`

(which takes a vector probability weights for obtaining the elements of the vector being sampled) to ensure that 30% *Young*, 40% *Adult* and 30% *Elderly* are sampled. Use the `table()` function to generate the following output (assigned to the variable `ans`). Also, show the proportions for each category.

```
# A summary of the sample (1000 elements),
# based on the probability weights
ans
#> pop
#>   Adult Elderly   Young
#>     399     300     301

# The proportions of each age
prop
#> pop
#>   Adult Elderly   Young
#>   0.399   0.300   0.301
```