Data Science for Operational Researchers Using R Online

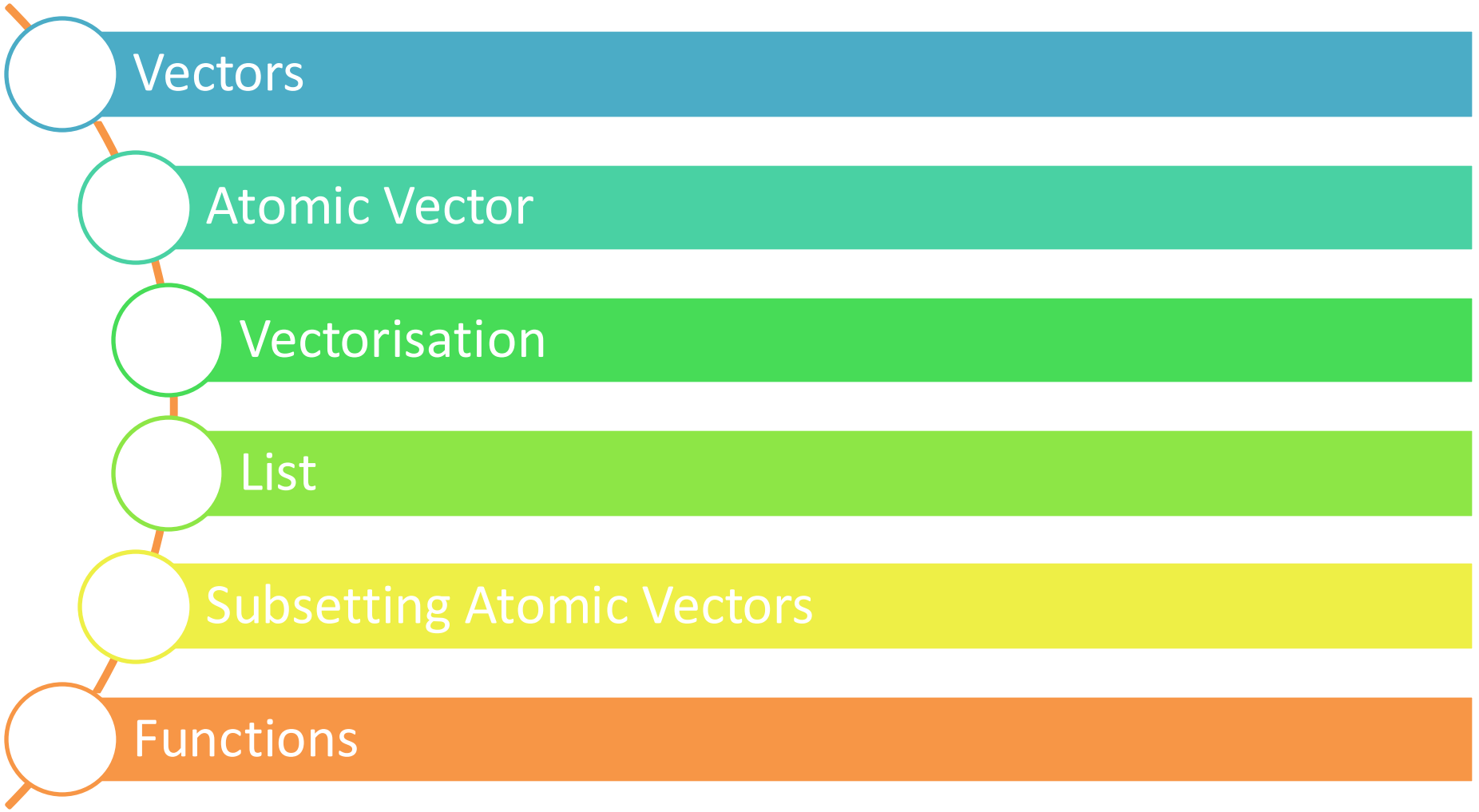
4. Vectors and Functions in R

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

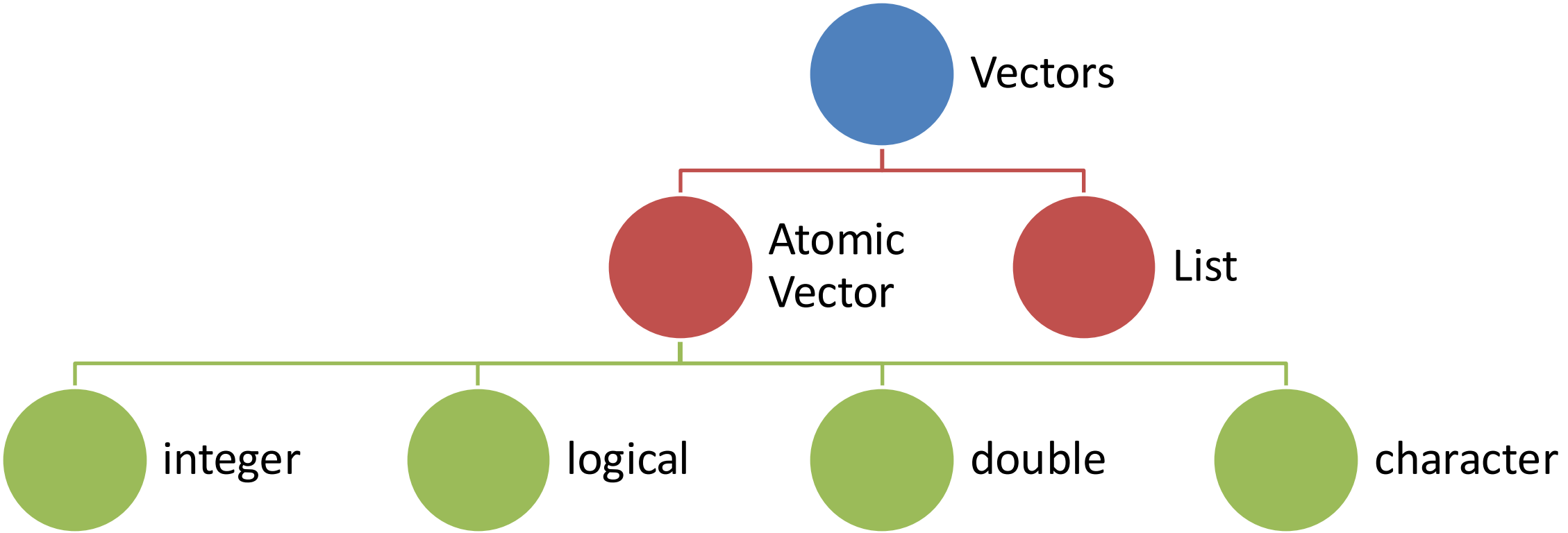
https://github.com/JimDuggan/explore_or

> 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)

Overview


Vectors
Atomic Vector
Vectorisation
List
Subsetting Atomic Vectors
Functions

1. Vectors


Vectors
Atomic Vector
List
integer
logical
double
character

2. 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**.

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.

```
# 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.

```
# 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
```

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

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 `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] "" "" ""
```

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
```

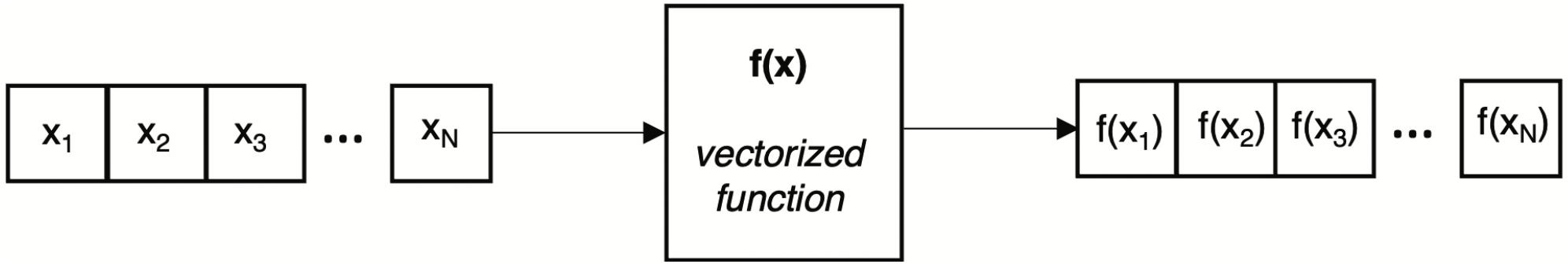
```
# Look for missing values in the vector v
is.na(v)
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

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

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

3. 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

```
# 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

```
# 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

```
# 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

```
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"
```

4. Lists

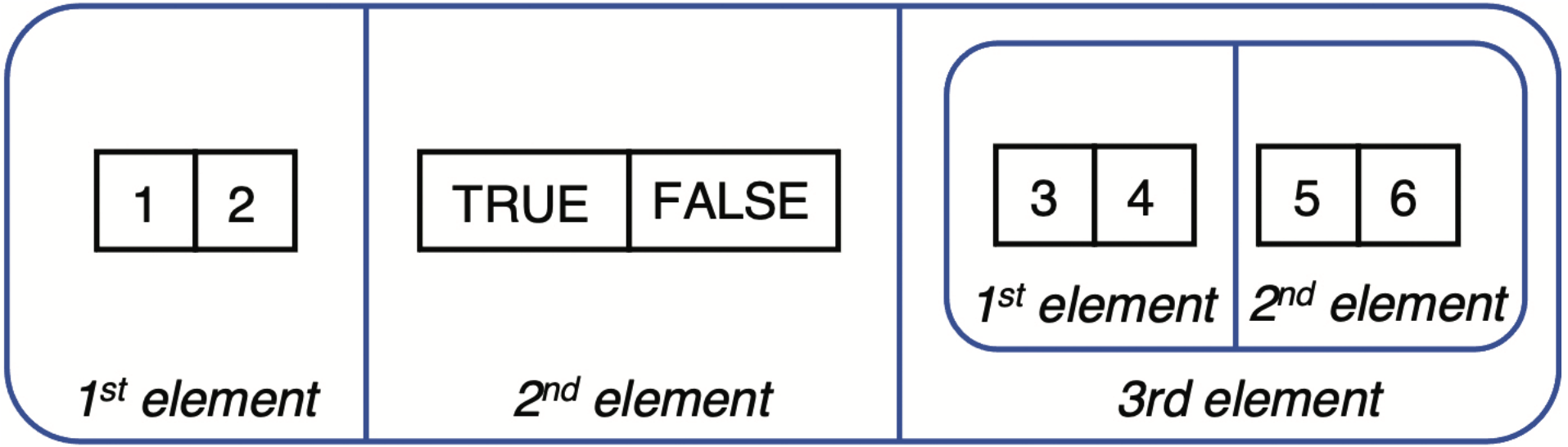
- A list is a vector that can contain different types, including a list.
- It is a flexible data structure and is often used to return data from a function.
- A list can be defined using the list() function, which is similar to the c() function used to create atomic vectors.

```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
```



Converting a list to an atomic vector - *flattening*

```
# 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"
```

5. Subsetting (atomic) vectors

> 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 (Wickham, 2019)

Overview

- Subsetting operators allow you to process data stored in atomic vectors and lists and R provides a range of flexible approaches that can be used to subset data.
- Methods
  - Positive Integer
  - Negative Integer
  - Logical Vector
  - Named Elements

```
# set the seed
set.seed(111)
# Generate the count data, assume a Poisson distribution
customers <- rpois(n = 10, lambda = 100)
# Name each successive element to be the day number
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
```

Note, subsetting lists involves more work than subsetting vectors, and we will not cover it here.

(a) Using positive integers

- Positive integers will subset atomic vector elements at given locations, and this type of index can have one or more values.
- To extract the n^{th} item from a vector x the term x[n] is used
- This can also apply to a sequence, for example, a sequence of values, starting at n and finishing 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 used to subset a vector.

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

```
# Get the customer from day 1
customers[1]
#>  D1
#> 102

# Get the customers from day 1 through day 5
customers[1:5]
#>  D1  D2  D3  D4  D5
#> 102  96  97  98 101

# Use c() to get the customers from day 1 and the final day
customers[c(1,length(customers))]
#>  D1 D10
#> 102  94
```

(b) Using negative integers

Negative integers can be used to exclude elements from a vector, and one or more elements can be excluded

```
# Exclude the first day's observation
customers[-1]
#>  D2  D3  D4  D5  D6  D7  D8  D9 D10
#>  96  97  98 101  85  98 118 102  94

# Exclude the first and last day
customers[-c(1,length(customers))]
#>  D2  D3  D4  D5  D6  D7  D8  D9
#>  96  97  98 101  85  98 118 102
```

(c) Using logical vectors

- Logical vectors can be used to subset a vector, and this is a powerful feature of R
- This allows for the use of relational and logical operators to perform subsetting.
- The idea is a simple one: when a logical vector is used to subset a vector, only the corresponding cells of the target vector that are TRUE in the logical vector will be retained.

For example, the code below will find any value that is greater than 100.

```
# Create a logical vector based on a relation expression
lv <- customers > 100
lv
#>    D1    D2    D3    D4    D5    D6    D7    D8    D9   D10
#>  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE

# Filter the original vector based on the logical vector
customers[lv]
#>  D1  D5  D8  D9
#> 102 101 118 102
```

Typically, these two statements are combined into the one expression, so you will often see the following style of code in R.

```
# Subset the vector to only show values great than 100
customers[customers > 100]
#>  D1  D5  D8  D9
#> 102 101 118 102
```

Recycling elements

- A convenient feature of subsetting with logical vectors is that the logical vector size does not have to equal the size of the target vector

```
# Subset every second element from the vector
customers[c(TRUE,FALSE)]
#>  D1  D3  D5  D7  D9
#> 102  97 101  98 102
```

(d) Using named elements

- If a vector has named elements, often set via the function names(), then elements can subsetted using their names.
- This is convenient if you need to retrieve an element but do not 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
```

which() function

```
# Use which() to find the indices of the true elements in the
# logical vector
v <- which(customers > 100)
v
#> D1 D5 D8 D9
#>  1  5  8  9

# Filter customers based on these indices
customers[v]
#>  D1  D5  D8  D9
#> 102 101 118 102
```

6. Functions in R

> An important advantage of R and other interactive languages is to make programming in the small an easy task. You should exploit this by creating functions habitually, as a natural reaction to any idea that seems likely to come up more than once.
>
> — John Chambers (Chambers, 2017)

Overview

- A function can be defined as a group of instructions that: takes input uses the input to compute other values, and returns a result (Matloff, 2011)
- Functions are declared using the function reserved word, and are objects, which means they can also be passed as arguments to other functions.

- The general form of a function in R is:
  - function (arguments) expression
  - arguments provides the arguments (inputs) to a function, and are separated by commas
  - expression is any legal R expression contained within the function body, and is usually enclosed in curly braces (when there is more than one expression),
  - the last evaluated expression is returned by the function, although the function return() can be also used to return values.

Removing duplicates

- We use an existing base R function to create a new one
- Our function will take in a vector of random numbers, and remove any duplicates
- To remove the duplicates, we will make use of the R function duplicated(), which returns a logical vector that contains TRUE if a value is duplicated
- We can invert this output to achieve our desired result.

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

```r
my_unique <- function(x){
  # Use duplicated() to create a logical vector
  dup_logi <- duplicated(x)
  # Invert the logical vector so that unique values 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
# The call to source loads the function into the global environment
source("my_functions.R")
```

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

Functions can be anonymous (no variable binding)

```
# Call my_summary() using an anonymous function
my_summary(1:10,function(y)min(y))
#> [1] 1
# Call my_summary() using an anonymous function
my_summary(1:10,function(y)max(y))
#> [1] 10
```

```
> 1:5 %>% (function(x)x-1)
[1] 0 1 2 3 4
```

Passing arguments to functions

- When programming in R, it is useful to distinguish between the fo*rmal 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 (e.g., by position)
- However, arguments can also be passed in by complete name or partial name, and arguments can have default values.

sum {base}

Sum of Vector Elements

Description

`sum` returns the sum of all the values present in its arguments.

Usage

```
sum(..., na.rm = FALSE)
```

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

Passing arguments…

- By position, where the arguments are copied to the corresponding argument location
- By complete name, where arguments are first copied to their corresponding name, before other arguments are then copied via their positions.
- By partial name, where argument names are matched, and where a unique match is found, that argument will be selected

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

```
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
```

Default values

- arguments can be allocated default values, which provides flexibility in that not all the arguments need to be called each time the function is invoked
- We can modify the function f so that each argument has an arbitrary default value.

```
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
```

General Guidelines for passing arguments

- Hadley Wickham (Wickham, 2019) provides valuable advice for passing arguments to functions, for example:
  1. to focus on positional mapping for the first one or two arguments
  2. to avoid positional mapping for arguments that are not used too often, and,
  3. unnamed arguments should come before named arguments.

Functionals – see purrrr content

- In R, functions are objects, so they can be passed to functions as arguments
- Functionals are functions that accept functions as arguments.
- To send a function as an argument, all that is required is the function name.

```
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
```

(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.

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
```