The first principle I propose is that our *Mission*, as users and creators of software for data analysis, is to enable the best and most thorough exploration of data possible. That means that users of the software must be able to ask meaningful questions about their applications, quickly and flexibly.

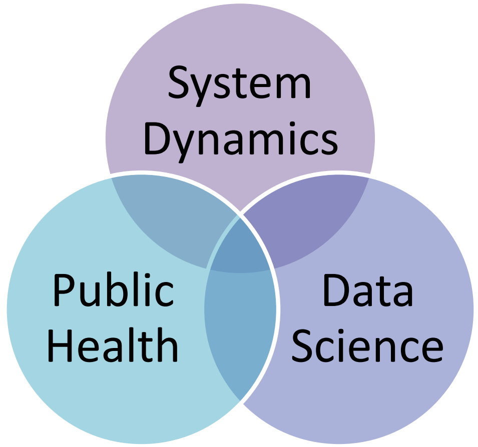
— John Chambers (Chambers, 2008)

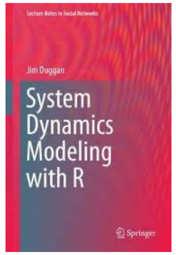
Programming for Data Analytics

01 – Atomic Vectors

Lecturer –Jim Duggan

- Lectures in
  - Programming (R, MATLAB),
  - Modelling & Simulation
- Research interests:
  - System Dynamics
  - Computational Epidemiology
  - Data Science





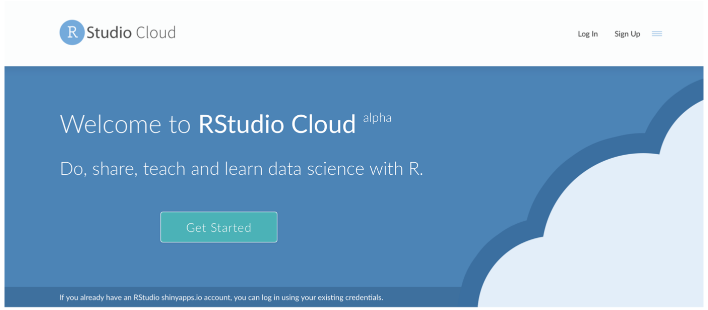
Insight Networks: Researchers from UL, University of Galway and TU Dublin partner to build on pandemic data insights

Course Structure

- Examination (End of Semester 1) – 60%
- Continuous Assessment – 40%
  - Weekly Quizzes (10%)
  - Lab Exam every ~3 weeks (supervised, in person) (30%)
- Topics
  - Base R (vectors, functions, S3)
  - tidyverse (ggplot2, dplyr, purrr)
  - Exploratory Data Analysis and RShiny

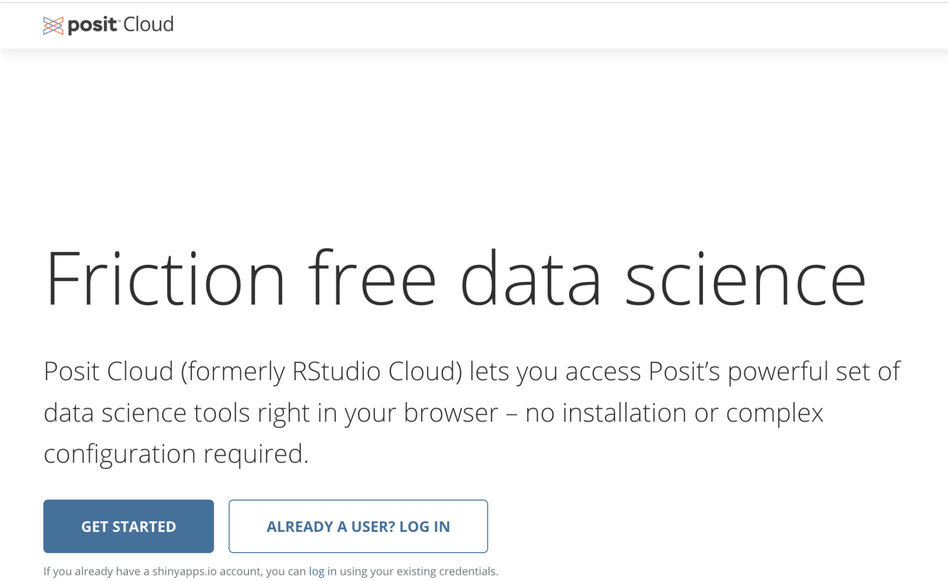
The R Project for Statistical Computing

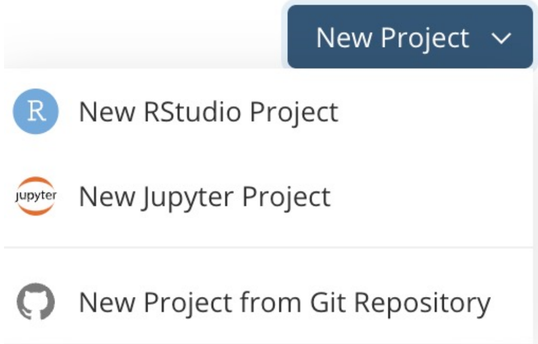
- R's *mission* is to enable the best and most thorough exploration of data possible (Chambers 2008).
- It is a dialect of the S language, developed at Bell Laboratories
- ACM noted that *S "will forever alter the way people analyze, visualize, and manipulate data"*

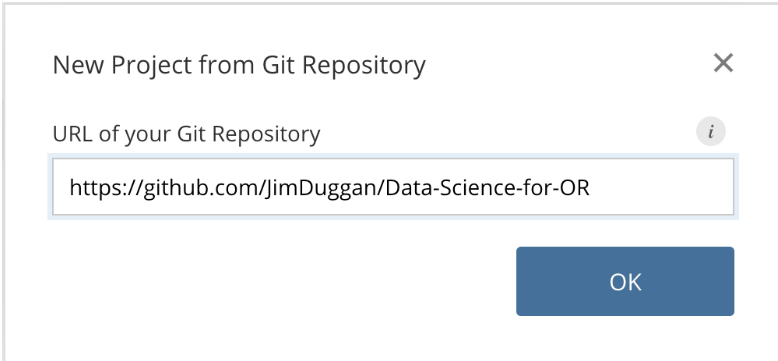



```r
# We use this for processing the answer
# In programming, we "stand on the shoulders of giants"
library(stringi)

# This gets the input from the user.
# The result is stored in a variable
# Variables are important in programming!
name <- readline(prompt="Enter a name: ")

# We call a specially designed function to get the answer
# In R, we call functions all the time
# A function is a "mini-program"
ans <- stri_reverse(name)

# After all this work, we output the result
cat("The reverse of ", name, "is ===>", ans)
```

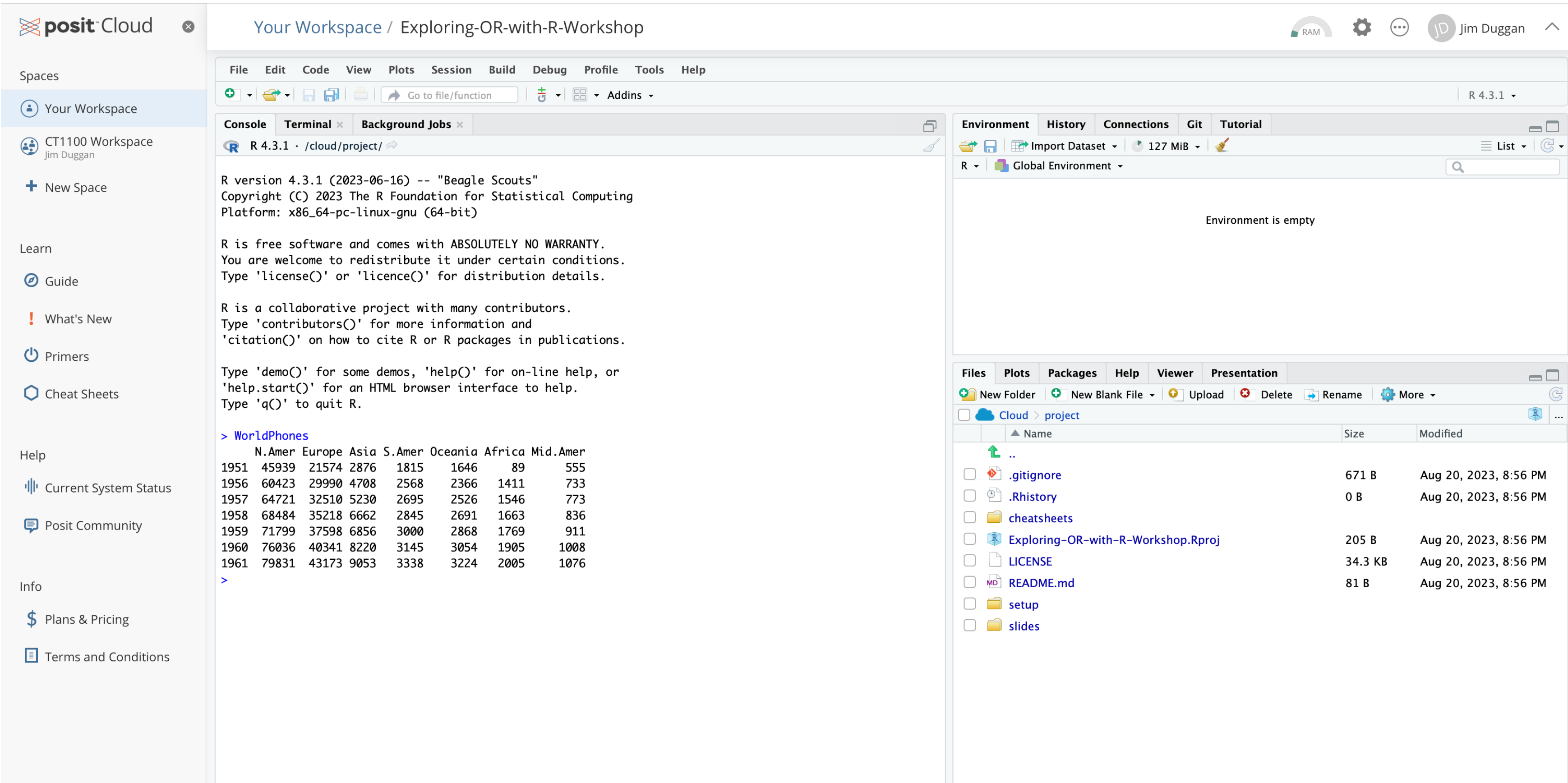
First Steps: Posit Cloud – Create Your Account



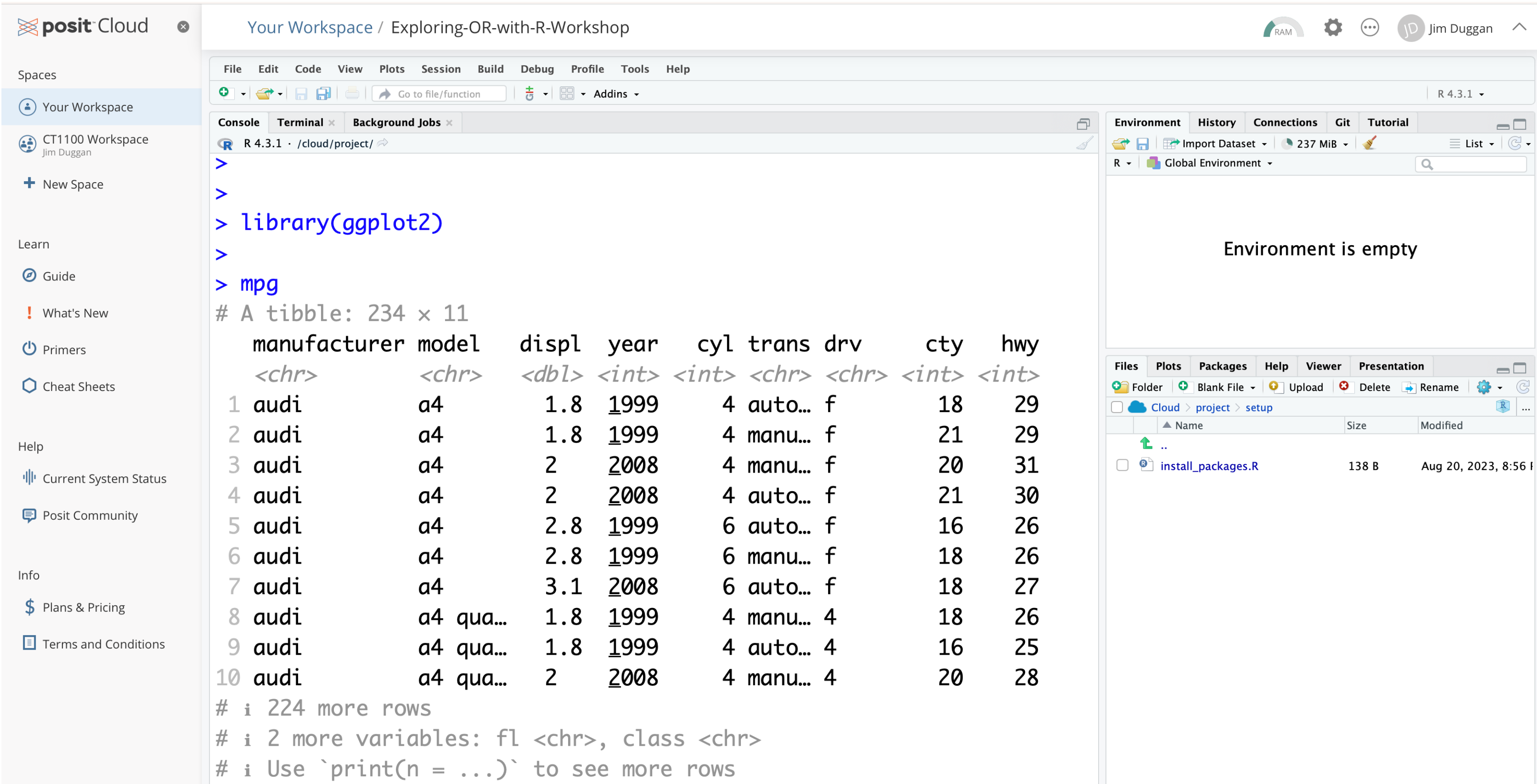




IDE Available for running scripts



A quick check… the ggplot2 **mpg** dataset



Atomic 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 (Matloff, 2011)

- A one-dimensional data structure that allows you to store one or more values.
- Created using the combine function c()
- Assignment using ← operator (convention in R)
- Four main types
  - logical
  - integer
  - numeric/double
  - character

```
# Create 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
```

Other types

```r
# 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

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

```r
# Create 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"
```

Combining 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

# Append for vector 4
v4 <- c(v2, v1)
v4
#> [1] 4 5 6 1 2 3
```

Creating large vectors

- Colon operator :
- seq() function
- rep() function
- vector() function

```
x <- 1:10

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

typeof(x)
#> [1] "integer"
length(x)
#> [1] 10
```

```
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=10,by=.5)
x2
#>  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0
#> [14]  7.5  8.0  8.5  9.0  9.5 10.0
```

vector() function

```r
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 vectors always contain data of the same type.

This is enforced by R using a process known as coercion.

Coercion

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

```
# Create a vector with integer and double combined
ex3 <- c(1L,2L, 3L, 4.1)
ex3
#> [1] 1.0 2.0 3.0 4.1
typeof(ex3)
#> [1] "double"
```

```
# Create a vector with logical, integer, double and character
# combined
ex4 <- c(TRUE,1L,2.0, "Hello")
ex4
#> [1] "TRUE"  "1"     "2"     "Hello"
typeof(ex4)
#> [1] "character"
```

Naming vector elements (useful)

```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

summary(x_dbl)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    1.20    3.40    7.20    7.12   11.10   12.70
```

```r
# 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

- When analysing data, it is common that there will be missing values
- A sensor (thermometer) might break down on any given day, and so an hourly temperature recording could be missed
- Logical constant NA is used in R to record a missing value ("Not Available")

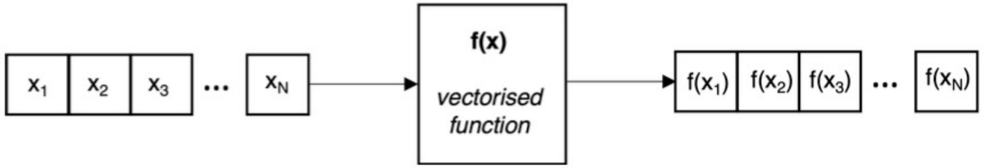
```
# define a vector v
v <- 1:10
v
#>  [1]  1  2  3  4  5  6  7  8  9 10

# 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

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

Detecting NAs

```
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
```

```
v
#>  [1]  1  2  3  4  5  6  7  8  9 NA
max(v, na.rm = TRUE)
#> [1] 9
```

Vectorisation



- Vectorisation is a powerful feature of R that where 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.
- In these scenarios, vectorisation removes the requirement to write loop structures that would iterate over the entire vector, and so leads to a simplified data analysis process.

```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)
v
#> [1] 10  7  6  3  1
length(v)
#> [1] 5
typeof(v)
#> [1] "integer"

# Call the vectorised function sqrt (square root)
rv <- sqrt(v)
rv
#> [1] 3.162 2.646 2.449 1.732 1.000
length(rv)
#> [1] 5
typeof(rv)
#> [1] "double"
```

R 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
# Vector subtraction
v1 - v2
#> [1]  8 16 27
```

R Relational Operators (Support Vectoristion)

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

Logical Operators

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

```r
# Setup a test vector, in this case, a sequence of random numbers
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

# Use logical OR to see which values are lower than 5 or greater than 17
v < 5 | v > 17
#>  [1] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE

# Use logical NOT to see which values are not even (using the remainder o
!(v %% 2 == 0)
#>  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

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"
```

Challenge 2.1

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] 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" "Medium"
#>  [8] "Hot"    "Medium" "Cold"   "Cold"   "Medium" "Medium" "Medium"
#> [15] "Medium" "Medium" "Hot"    "Medium" "Cold"   "Medium"
```

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 (Wickham, 2019)

- Subsetting operations allow you to process data stored in atomic vectors, and R provides a range of flexible approaches that can be used to subset data
- There are 4 ways:
  - Positive integer
  - Negative integer
  - Logical vectors
  - Named elements

```r
# set the seed
set.seed(111)
```

```r
# 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
```

Positive integers

```
# Get the customer from day 1
customers[1]
#>  D1
#> 102
# Get the customers from day 1 through to 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
# Note, with c(), any duplicates will be returned
customers[c(1:3,3,3)]
#>  D1  D2  D3  D3  D3
#> 102  96  97  97  97
```

Negative integers (exclusion)

```
# 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
# Exclude all values except the first and last day
customers[-(2:(length(customers)-1))]
#>  D1 D10
#> 102  94
```

Logical vectors (1)

```
# 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
```

Logical vectors (2)

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

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

Named elements

```
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
```

```
# Extract the first and last elements
customers[c("D1","D10")]
#>  D1 D10
#> 102  94
```

Mini-Case for Atomic Vectors – Simulating Dice Rolls

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

```
> sample(1:10,10,replace = T)
 [1]  6  6 10  5  4  4 10  8  3 10
>
>
> sample(1:10,10)
 [1] 10  5  4  1  6  3  9  8  7  2
```

Random Samples and Permutations

Description

`sample` takes a sample of the specified size from the elements of `x` using either with or without replacement.

Usage

```
sample(x, size, replace = FALSE, prob = NULL)
```

```
> sample(c("YES","NO"),100,replace = T,prob=c(0.1,0.9))
  [1] "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "YES" "NO"  "NO" 
 [12] "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO" 
 [23] "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO" 
 [34] "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "YES" "NO" 
 [45] "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "YES" "NO" 
 [56] "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO" 
 [67] "NO"  "NO"  "YES" "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "NO" 
 [78] "NO"  "NO"  "NO"  "NO"  "NO"  "NO"  "YES" "NO"  "NO"  "NO"  "NO" 
 [89] "NO"  "NO"  "NO"  "NO"  "YES" "NO"  "NO"  "NO"  "NO"  "NO"  "NO" 
[100] "NO" 
```

```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
head(dice2)
#> [1] 4 5 2 5 4 2
summary(dice1)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    1.00    2.00    3.00    3.49    5.00    6.00
```

```r
# Create a new variable dice_sum, a vectorised 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, converted to an atomic vector
freq <- as.vector(table(dice_sum))
freq
#>  [1]  274  569  833 1070 1387 1687 1377 1165  807  534  297
```

```r
# Show the frequency proportions for the summed values,
# using the vectorised division operator
sim_probs <- freq/length(dice_sum)
sim_probs
#>  [1] 0.0274 0.0569 0.0833 0.1070 0.1387 0.1687 0.1377 0.1165 0.0807
#> [10] 0.0534 0.0297

# Define the exact probabilities for the sum of two dice throws
exact <- c(1/36, 2/36, 3/36, 4/36, 5/36, 6/36, 5/36, 4/36, 3/36, 2/36, 1/36)

# Use vectorised subtraction to show the differences,
# rounded to 5 decimal places.
round(sim_probs - exact, 5)
#>  [1] -0.00038  0.00134 -0.00003 -0.00411 -0.00019  0.00203 -0.00119
#>  [8]  0.00539 -0.00263 -0.00216  0.00192
```

R Function	Description
`as.vector()`	Coerces its argument into a vector
`c()`	Used to create an atomic vector, with elements separated by commas
`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 vectors
`list()`	A function to construct a list
`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 names
`str()`	Compactly displays the internal structure of a variable
`set.seed()`	Provides a way to initialize a pseudorandom number generator.
`sample()`	Generates a random sample of values, with or without replacement

R Function	Description
`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