1.2.1 Basic Concepts in R

Hello World Program

Write the “Hello World” program using both (1) R command prompt and (2) writing a script.

R Command Prompt

Once you have R environment setup, then it’s easy to start your R command prompt by just typing the following command at your command prompt:

```
$ R
```

This will launch R interpreter and you will get a prompt > where you can start typing your program as follows:

```
        myString <- "Hello, World!"
        print (myString)
```

```
[1] "Hello, World!"
```

Here first statement defines a string variable myString, where we assign a string “Hello, World!” and then next statement print() is being used to print the value stored in variable myString.

Note
The bracketed number [1] that appears before the output indicates that the line begins with the first value of the result. Some results may have multiple values that fill several lines, so this indicator is occasionally useful but can generally be ignored.

Writing R Script File

Usually, you will do your programming by writing your programs in script files and then you execute those scripts at your command prompt with the help of R interpreter called Rscript.

So let’s start with writing following code in a text file called test.R:

```
        # My first program in R Programming
        myString <- "Hello, World!"
        print ( myString)
```

```
[1] "Hello, World!"
```

Save the above code in a file test.R and execute it at Linux command prompt as given below.

Even if you are using Windows or other system, syntax will remain same.

```
$ Rscript test.R
```

Comments

Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program. Single comment is written using # in the beginning of the statement as follows:

```
        # My first program in R Programming
```

Variables and Basic Data Types

- Generally, while doing programming in any programming language, you need to use various variables to store various information.
- Variables are reserved memory locations to store values. This means that, when you create a variable you reserve some space in memory.
- You may need to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc.
- Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.
- In contrast to other programming languages like C and java in R, the variables are not declared as some data type. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects.
- The frequently used ones are − ***Vectors Lists - Matrices - Arrays - Factors - Data Frames***

Numeric Variables

Numeric variables are the most common type of variable and are used to store real numbers.

- **Integer**: A subtype of numeric that represents whole numbers.

```
        # Integer variable
        count <- 42L # The 'L' specifies it as an integer

        # Floating-point number
        weight <- 72.5
```

- **Complex Numbers**: Numbers with a real and an imaginary part.

```
# Complex number
complex_num <- 3+4i
```

- **Character Variables**

Character variables store text and are enclosed in quotes.

```
# Single character string
name <- "Alice"

# Multiple characters string
sentence <- "Hello, World!"
```

- **Logical Variables**

Logical variables can only take two values: TRUE or FALSE. They are the result of conditions or logical operations.

```
# Logical variable
is_valid <- TRUE

# Result of a logical operation
test_result <- 5 > 3 # Returns TRUE

a <- T
b <- F

a | b
```

```
[1] TRUE
```

```
a & b
```

```
[1] FALSE
```

R is Case Sensitive

The R language is case-sensitive, so typing the command as Print( ) or PRINT( ) will simply produce an error.

```
age <- 25
print(age)
```

```
[1] 25
```

```
print(Age)
```

```
Error in eval(expr, envir, enclos): object 'Age' not found
```

```r
Print(age)
```

```
Error in Print(age): could not find function "Print"
```

Get Data Type of a Variable

```r
BMI <- data.frame(
gender = c("Male", "Male","Female"),
height = c(152, 171.5, 165),
weight = c(81,93, 78),
Age = c(42,38,26)
)
```

```r
print("The class of BMI is:")
```

```
[1] "The class of BMI is:"
```

```r
class(BMI)
```

```
[1] "data.frame"
```

```r
print("The type of BMI is:")
```

```
[1] "The type of BMI is:"
```

```r
typeof(BMI)
```

```
[1] "list"
```

```r
print("The structure of BMI is:")
```

```
[1] "The structure of BMI is:"
```

```r
str(BMI)
```

```
'data.frame':   3 obs. of  4 variables:
 $ gender: chr  "Male" "Male" "Female"
 $ height: num  152 172 165
 $ weight: num  81 93 78
 $ Age   : num  42 38 26
```

Logical operators in R:

<	less than
>	great than
<=	less than or equal
>=	greater than or equal
==	equal to
!=	not equal to
|	or
!	not
&	and

Complext Data Types and Data Structures

Vectors

Vectors are one-dimensional arrays that can hold numeric, character, or logical values, but all elements must be of the same type.

Patient ID	Height (CM)	Weight (KG)	Systolic (mmHg)	Diastolic (mmHg)
10200	165	71	120	80
3600	180	92	110	85
8437	172	76	118	82
5911	167	82	115	78

```r
# Numeric vector
ages <- c(25, 30, 45)

# Character vector
colors <- c("red", "green", "blue")

# Logical vector
answers <- c(TRUE, FALSE, TRUE)
```

Creating Sequences in R

Sequences are ordered sets of numbers that follow a specific pattern.

In R, sequences are commonly used for tasks such as iteration in loops, subsetting data structures, and during plotting.

R provides several functions to create sequences efficiently.

Using the Colon Operator `:`

The colon operator is the simplest way to create a sequence of numbers in R. It generates a sequence from a starting point to an endpoint with a step of 1.

Syntax:

```
start:end
```

Example:

Create a sequence from 1 to 10.

```
seq_1_to_10 <- 1:10
print(seq_1_to_10)
```

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

The `seq()` Function

The `seq()` function is more versatile than the colon operator and can create sequences with specified increments (the `by` argument) or of a specified length (the `length.out` argument).

Syntax:

```
seq(from, to, by, length.out)`
```

Examples:

- Create a sequence from 1 to 10 with a step of 2:

```
seq_by_2 <- seq(from = 1, to = 10, by = 2)
print(seq_by_2)
```

```
[1] 1 3 5 7 9
```

- Create a sequence from 5 to 10, consisting of 3 equally spaced numbers:
-

```
seq_length_out <- seq(from = 5, to = 10, length.out = 3)
print(seq_length_out)
```

```
[1]  5.0  7.5 10.0
```

The `rep()` Function

The `rep()` function is used to replicate the values in x. It is a powerful function for repeating sequences.

Syntax:

```
rep(x, times, each, length.out)
```

Examples:

- Repeat the sequence of numbers 1, 2, 3, three times:

```
rep_times <- rep(x = 1:3, times = 3)
print(rep_times)
```

```
[1] 1 2 3 1 2 3 1 2 3
```

- Repeat each number in the sequence 1, 2, 3, three times:

```
rep_each <- rep(x = 1:3, each = 3)
print(rep_each)
```

```
[1] 1 1 1 2 2 2 3 3 3
```

Combining Sequences

Sequences can be combined using the `c()` function to create longer and more complex sequences.

Example:

Combine two sequences into one.

```
combined_seq <- c(seq(1, 5), seq(8, 10))
print(combined_seq)
```

```
[1]  1  2  3  4  5  8  9 10
```

Practice Vectors

- Create a vector that combines the numbers 1, 2, and 3 with the letters 'a', 'b', and 'c'.
- Create a vector containing the numbers 1 to 10, but only include every second number.
- Create a vector that contains the sequence 1, 2, 3, followed by the sequence 4, 5, 6, each sequence repeated twice.
- Create a vector that starts with the sequence 1, 2, 3 and then repeats the number 4 five times.
- Create a vector containing the first 10 even numbers.
- Create a vector containing the first 10 odd numbers.

- Create a vector that uses the `seq()` function to generate numbers from 1 to 10, but each number should appear twice.
- Create a vector that uses the `rep()` function to repeat the sequence 1, 2, 3, but the entire sequence should appear three times.

Accessing Elements in a Sequence Data Type e.g. Vector, Data Frame

Access Vector Elements

Accessing elements from a vector in R is straightforward and can be achieved using indexing with square brackets `[ ]`.

You can specify the index or indices of the elements you want to extract. Remember, R is 1-based indexing, meaning the first element of a vector has an index of 1.

Single Element Access

To access a single element from a vector, you provide its index within square brackets immediately following the vector name.

```r
# Creating a numeric vector
numbers <- c(10, 20, 30, 40, 50)

# Access the third element
third_number <- numbers[3]
print(third_number)  # Outputs 30
```

```
[1] 30
```

Multiple Elements Access

To access multiple elements, you can pass a vector of indices inside the square brackets.

```r
# Access the first, third, and fifth elements
selected_numbers <- numbers[c(1, 3, 5)]
print(selected_numbers)  # Outputs 10, 30, 50
```

```
[1] 10 30 50
```

Sequential Elements Access

R allows you to use the colon `:` operator to create a sequence, which can be used for accessing a range of elements.

```r
# Access the second to the fourth elements
range_numbers <- numbers[2:4]
print(range_numbers)  # Outputs 20, 30, 40
```

```
[1] 20 30 40
```

Conditional Access

You can also access elements based on conditions. This method is useful for filtering elements.

```r
# Access elements greater than 25
filtered_numbers <- numbers[numbers > 25]
print(filtered_numbers)  # Outputs 30, 40, 50
```

```
[1] 30 40 50
```

Find the Index of Matching Elements

```r
which(numbers > 25)
```

```
[1] 3 4 5
```

Excluding Elements

Adding a minus sign `-` before an index or a vector of indices tells R to exclude those elements.

```r
# Exclude the second element
exclude_second <- numbers[-2]
print(exclude_second)  # Outputs 10, 30, 40, 50
```

```
[1] 10 30 40 50
```

```r
# Exclude second to fourth elements
exclude_range <- numbers[-(2:4)]
print(exclude_range)  # Outputs 10, 50
```

```
[1] 10 50
```

Named Vectors

For named vectors, you can access elements using their names in a similar manner.

```r
# Creating a named vector
fruits <- c(apple = 10, banana = 5, cherry = 8)

# Access the element named 'banana'
banana_count <- fruits["banana"]
print(banana_count)  # Outputs 5
```

```
banana
     5
```

```r
# Access multiple elements by name
some_fruits <- fruits[c("apple", "cherry")]
print(some_fruits)  # Outputs apple 10, cherry 8
```

```
 apple cherry
    10      8
```

Basic Operations on Vectors

- Once you have a vector (or a list of numbers) in memory most basic operations are available.
- Most of the basic operations will act on a whole vector and can be used to quickly perform a large number of calculations with a single command.
- There is one thing to note, if you perform an operation on more than one vector it is often necessary that the vectors all contain the same number of entries.
- Here we first define a vector which we will call “a” and will look at how to add and subtract constant numbers from all of the numbers in the vector.
- First, the vector will contain the numbers 1, 2, 3, and 4.
- We then see how to add 5 to each of the numbers, subtract 10 from each of the numbers, multiply each number by 4, and divide each number by 5.

```r
a <- c(1,2,3,4)
a
```

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

```r
a <- a + 5
a
```

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

```r
a <- a - 10
a
```

```
[1] -4 -3 -2 -1
```

```r
a <- a / 5
a
```

```
[1] -0.8 -0.6 -0.4 -0.2
```

We can save the results in another vector called b:

```
b <- a
b
```

```
[1] -0.8 -0.6 -0.4 -0.2
```

If you want to raise to a power, take the square root, find e raised to each number, the logarithm, etc., then the usual commands can be used:

```
a <- c(1,2,3,4)
a ^ 2
```

```
[1]  1  4  9 16
```

```
sqrt(a)
```

```
[1] 1.000000 1.414214 1.732051 2.000000
```

```
exp(a)
```

```
[1]  2.718282  7.389056 20.085537 54.598150
```

```
log(a)
```

```
[1] 0.0000000 0.6931472 1.0986123 1.3862944
```

```
exp(log(a))
```

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

By combining operations and using parentheses you can make more complicated expressions:

```
c <- (a + sqrt(a))/(exp(2)+1)
c
```

```
[1] 0.2384058 0.4069842 0.5640743 0.7152175
```

Note that you can do the same operations with vector arguments.

For example to add the elements in vector a to the elements in vector b use the following command:

```
a + b
```

```
[1] 0.2 1.4 2.6 3.8
```

The operation is performed on an element by element basis.

Note this is true for almost all of the basic functions.

So you can bring together all kinds of complicated expressions:

```
a*b
```

```
[1] -0.8 -1.2 -1.2 -0.8
```

```
a/b
```

```
[1]  -1.250000  -3.333333  -7.500000 -20.000000
```

```
(a+3)/(sqrt(1-b)*2-1)
```

```
[1] 2.376311 3.268354 4.390998 5.877956
```

You need to be careful of one thing. When you do operations on vectors they are performed on an element by element basis. One ramification of this is that all of the vectors in an expression must be the same length. If the lengths of the vectors differ then you may get an error message, or worse, a warning message and unpredictable results:

```
a <- c(1,2,3)
b <- c(10,11,12,13)
a+b
```

Basic Numerical Descriptors

The following commands can be used to get the mean, median, quantiles, minimum, maximum, variance, and standard deviation of a set of numbers:

```
a <- 1:10
mean(a)
```

```
[1] 5.5
```

```
median(a)
```

```
[1] 5.5
```

```
quantile(a)
```

```
   0%   25%   50%   75%  100% 
 1.00  3.25  5.50  7.75 10.00 
```

```
var(a)
```

```
[1] 9.166667
```

```r
sd(a)
```

```
[1] 3.02765
```

```r
min(a)
```

```
[1] 1
```

```r
max(a)
```

```
[1] 10
```

Finally, the summary command will print out the min, max, mean, median, and quantiles:

```r
summary(a)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   1.00    3.25    5.50    5.50    7.75   10.00 
```

The summary command is especially nice because if you give it a data frame it will print out the summary for every vector in the data frame:

```r
summary(mtcars)
```

```
      mpg             cyl             disp             hp       
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
 Median :19.20   Median :6.000   Median :196.3   Median :123.0  
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
      drat             wt             qsec             vs        
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
       am              gear            carb      
 Min.   :0.0000   Min.   :3.000   Min.   :1.000  
 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
 Median :0.0000   Median :4.000   Median :2.000  
 Mean   :0.4062   Mean   :3.688   Mean   :2.812  
 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
 Max.   :1.0000   Max.   :5.000   Max.   :8.000  
```

Sorting Vectors

Here we look at some commonly used commands that perform operations on lists.

The commands include the sort, min, max, and sum commands.

First, the sort command can sort the given vector in either ascending or descending order:

```r
a = c(2,4,6,3,1,5)
b = sort(a)
c = sort(a,decreasing = TRUE)
```

Speed of Vectorized Operations

Using vectorization to multiply numbers vs using for loop

```
a <- rnorm(N)
b <- rnorm(N)

# Vectorized approach
c <- a * b

# De-vectorized approach
d <- rep(NA, N)
for(i in 1:N){
  d[i] <- a[i] * b[i]
}
```

```r
N <- 100000000
a <- rnorm(N)
b <- rnorm(N)

# Measure the time for the vectorized approach
vectorized_time <- system.time({
  c <- a * b
})

# Measure the time for the de-vectorized approach
de_vectorized_time <- system.time({
  d <- rep(NA, N)
  for(i in 1:N){
    d[i] <- a[i] * b[i]
  }
})

# Print the timings
print(vectorized_time)
```

```
   user  system elapsed 
   0.03    0.00    0.17 
```

```r
print(de_vectorized_time)
```

```
   user  system elapsed 
   1.22    0.03    2.96 
```

Other Data Structures (Matrix, Array, List and Data frames)

Matrices

Matrices are two-dimensional arrays that contain elements of the same type. They have rows and columns.

```r
# Matrix with 3 rows and 2 columns
matrix_data <- matrix(1:6, nrow = 3, ncol = 2)
```

Note: Check the NBA Example Quarto File

Arrays

Arrays are similar to matrices but can have more than two dimensions.

```r
# 3-dimensional array
array_data <- array(1:24, dim = c(3, 4, 2))
```

Lists

Lists can hold elements of different types, including numbers, strings, vectors, and even other lists.

```r
# List containing different types
list_data <- list(name = "Alice", age = 25, scores = c(90, 80, 85))

print(list_data[1])
```

```
$name
[1] "Alice"
```

Data Frames

Data frames are used to store tabular data. They are similar to matrices but can contain different types of variables.

```r
# Data frame
df <- data.frame(
  Name = c("Alice", "Bob"),
  Age = c(25, 30),
  Weight = c(55.5, 85.7)
)
```

Access Data Frame Elements

R built-in datasets

R provides several datasets that can be used for practice.

Use the data() command to examine these datasets.

```
#data()

# we will use mtcars dataset for demonstration
#?mtcars
#View(trees)
```

Using $ to Access Columns

The `$` operator is used to access a column directly by its name, returning the column as a vector.

```
miles_per_hour <- mtcars$mpg
print(miles_per_hour)
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

Using [] for Row and Column Indexing

The `[ ]` operator allows for more flexible sub-setting, including selecting rows, columns, or both, using numeric indexes or column names.

- **Select a single column by name**:

```
miles_per_hour <- mtcars["mpg"]
print(miles_per_hour)
```

```
                     mpg
Mazda RX4           21.0
Mazda RX4 Wag       21.0
Datsun 710          22.8
Hornet 4 Drive      21.4
Hornet Sportabout   18.7
Valiant             18.1
Duster 360          14.3
Merc 240D           24.4
Merc 230            22.8
Merc 280            19.2
Merc 280C           17.8
Merc 450SE          16.4
Merc 450SL          17.3
Merc 450SLC         15.2
Cadillac Fleetwood  10.4
Lincoln Continental 10.4
Chrysler Imperial   14.7
Fiat 128            32.4
Honda Civic         30.4
Toyota Corolla      33.9
Toyota Corona       21.5
Dodge Challenger    15.5
AMC Javelin         15.2
Camaro Z28          13.3
Pontiac Firebird    19.2
Fiat X1-9           27.3
Porsche 914-2       26.0
Lotus Europa        30.4
Ford Pantera L      15.8
Ferrari Dino        19.7
Maserati Bora       15.0
Volvo 142E          21.4
```

Select multiple columns by name

```r
subset_df <- mtcars[c("mpg", "cyl")]
print(subset_df)
```

```
                     mpg cyl
Mazda RX4           21.0   6
Mazda RX4 Wag       21.0   6
Datsun 710          22.8   4
Hornet 4 Drive      21.4   6
Hornet Sportabout   18.7   8
Valiant             18.1   6
Duster 360          14.3   8
Merc 240D           24.4   4
Merc 230            22.8   4
Merc 280            19.2   6
Merc 280C           17.8   6
Merc 450SE          16.4   8
Merc 450SL          17.3   8
Merc 450SLC         15.2   8
Cadillac Fleetwood  10.4   8
Lincoln Continental 10.4   8
Chrysler Imperial   14.7   8
Fiat 128            32.4   4
Honda Civic         30.4   4
Toyota Corolla      33.9   4
Toyota Corona       21.5   4
Dodge Challenger    15.5   8
AMC Javelin         15.2   8
Camaro Z28          13.3   8
Pontiac Firebird    19.2   8
```

```
Fiat X1-9           27.3   4
Porsche 914-2       26.0   4
Lotus Europa        30.4   4
Ford Pantera L      15.8   8
Ferrari Dino        19.7   6
Maserati Bora       15.0   8
Volvo 142E          21.4   4
```

```
# In R, rownames is a function that retrieves or sets the row names of a matrix or # dat
# rownames are essentially a character vector of unique identifiers that label the rows,

# Get all rownames
car_names <- rownames(mtcars)
print(car_names)
```

```
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"         
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"            
 [7] "Duster 360"          "Merc 240D"           "Merc 230"           
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"         
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood" 
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"           
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"      
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"         
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"      
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"       
[31] "Maserati Bora"       "Volvo 142E"
```

```
# Get first rowname only
first_car_name <- rownames(mtcars)[1]
print(first_car_name)
```

```
[1] "Mazda RX4"
```

```
# Find record by rowname
datsun_710 <- mtcars["Datsun 710", ]
print(datsun_710)
```

```
            mpg cyl disp hp drat   wt  qsec vs am gear carb
Datsun 710 22.8   4  108 93 3.85 2.32 18.61  1  1    4    1
```

```
mtcars$car_brand <- rownames(mtcars)
rownames(mtcars) <- NULL
head(mtcars)
```

```
   mpg cyl disp  hp drat    wt  qsec vs am gear carb         car_brand
1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4         Mazda RX4
2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4     Mazda RX4 Wag
```

```
3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1        Datsun 710
4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1    Hornet 4 Drive
5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2 Hornet Sportabout
6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1           Valiant
```

```
#View(mtcars)
```

Select a single row

```
first_row <- mtcars[1, ]
print(first_row)
```

```
  mpg cyl disp  hp drat   wt  qsec vs am gear carb car_brand
1  21   6  160 110  3.9 2.62 16.46  0  1    4    4 Mazda RX4
```

Select a specific element

```
# Accessing the Age of the second person
mpg <- mtcars[2, "mpg"]
print(mpg)
```

```
[1] 21
```

Using `subset()` Function

The `subset()` function can be used for more complex subsetting based on conditions.

- **Select rows where Age is greater than 25**:

  ```
  more_than_4_cyl <- subset(mtcars, cyl > 4)
  print(more_than_4_cyl)
  ```

```
    mpg cyl  disp  hp drat    wt  qsec vs am gear carb           car_brand
1  21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4           Mazda RX4
2  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4       Mazda RX4 Wag
4  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1      Hornet 4 Drive
5  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2   Hornet Sportabout
6  18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1             Valiant
7  14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4          Duster 360
10 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4            Merc 280
11 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4           Merc 280C
12 16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3          Merc 450SE
13 17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3          Merc 450SL
14 15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3         Merc 450SLC
15 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4  Cadillac Fleetwood
16 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4 Lincoln Continental
17 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4   Chrysler Imperial
22 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2    Dodge Challenger
23 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2         AMC Javelin
```

```
    24 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4         Camaro Z28
    25 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2   Pontiac Firebird
    29 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4     Ford Pantera L
    30 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6       Ferrari Dino
    31 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8      Maserati Bora
```

Filtering data frames

```r
# Example of creating a data.frame
name <- c("John", "Alice", "Bob", "Eva")
age <- c(23, 25, 30, 22)
gender <- c("Male", "Female", "Male", "Female")

# Combine vectors into a data.frame
students <- data.frame(name, age, gender)

# Display the data.frame
print(students)
```

```
   name age gender
1  John  23   Male
2 Alice  25 Female
3   Bob  30   Male
4   Eva  22 Female
```

```r
# Accessing a column using $
print(students$age)
```

```
[1] 23 25 30 22
```

```r
# Accessing a column using indexing
print(students["gender"])
```

```
  gender
1   Male
2 Female
3   Male
4 Female
```

```r
# Access the element in the 2nd row, 1st column
print(students[2, 1])
```

```
[1] "Alice"
```

```r
# Access the entire 3rd row
print(students[3, ])
```

```
  name age gender
3  Bob  30   Male
```

```
# Access the entire 2nd column
print(students[, 2])
```

```
[1] 23 25 30 22
```

```
# Filter rows where age is greater than 23
older_students <- students[students$age > 23, ]

# Display the filtered data.frame
print(older_students)
```

```
   name age gender
2 Alice  25 Female
3   Bob  30   Male
```

Miscellaneous Related Topics - مواضيع متفرقة ذات علاقة

Printing Output

In R, you have several functions at your disposal to print output. Here are the most commonly used ones:

1. **`print()`** - This is the basic R function to print objects to the console. It's a generic function, which means that it can have different methods for different object types.

   ```
   print("Hello, World!")
   ```

   ```
   [1] "Hello, World!"
   ```

2. **`cat()`** - Concatenates and prints objects. It is often used for printing character strings and does not automatically add a newline unless you specify it with `"\n"`.

   ```
   cat("Hello,", "World!", "\n")
   ```

   ```
   Hello, World!
   ```

3. The **`paste`** function in R is used to concatenate strings. Unlike `print` or `cat`, `paste` does not print to the console but returns a single string that is the concatenation of its arguments. The `paste` function is particularly useful when you want to combine strings and variables to create a new string.

Here's a basic usage of `paste`:

```
# This will not print to the console but will return a string.
result <- paste("Hello", "World", sep = " ")
```

```r
# To print the result, you can use print() or cat()
print(result)  # This will print "Hello World" to the console
```

```
[1] "Hello World"
```

```r
cat(result, "\n")  # This will also print "Hello World", followed by a new line
```

```
Hello World
```

Requesting user input

Test this command in the console window not in quarto.

```r
name <- readline("Please enter your name: ")
cat("Your name is:", name)
```

Values Assignments in R

We can use different syntax to make an assignment in R.

The preferred syntax is to use the Values Gets e.g. x <- 10, which means x Gets the value 10.

We can also use the conventional assignment operator.

The Gets operator has the advantage of escaping the ambiguity of the assignment and equality operators = vs ==

```r
x <- 1 # valid, x gets 1
x
```

```
[1] 1
```

```r
1 -> x # valid, x gets 1
x
```

```
[1] 1
```

```r
x = 1 # valid, x = 1
x
```

```
[1] 1
```

```r
1 = x # invalid, x
```

```
Error in 1 = x: invalid (do_set) left-hand side to assignment
```

List Memory, Clean Memory

```
# list variables
x <- 1
y <- 2
ls()
```

```
 [1] "a"                 "age"               "ages"             
 [4] "answers"           "array_data"        "b"                
 [7] "banana_count"      "BMI"               "c"                
[10] "car_names"         "colors"            "combined_seq"     
[13] "complex_num"       "count"             "d"                
[16] "datsun_710"        "de_vectorized_time" "df"               
[19] "exclude_range"     "exclude_second"    "filtered_numbers" 
[22] "first_car_name"    "first_row"         "fruits"           
[25] "gender"            "i"                 "is_valid"         
[28] "list_data"         "matrix_data"       "miles_per_hour"   
[31] "more_than_4_cyl"   "mpg"               "mtcars"           
[34] "myString"          "N"                 "name"             
[37] "numbers"           "older_students"    "range_numbers"    
[40] "rep_each"          "rep_times"         "result"           
[43] "selected_numbers"  "sentence"          "seq_1_to_10"      
[46] "seq_by_2"          "seq_length_out"    "some_fruits"      
[49] "students"          "subset_df"         "test_result"      
[52] "third_number"      "vectorized_time"   "weight"           
[55] "x"                 "y"                
```

```
# clear specific values
rm(x,y)
ls()
```

```
 [1] "a"                 "age"               "ages"             
 [4] "answers"           "array_data"        "b"                
 [7] "banana_count"      "BMI"               "c"                
[10] "car_names"         "colors"            "combined_seq"     
[13] "complex_num"       "count"             "d"                
[16] "datsun_710"        "de_vectorized_time" "df"               
[19] "exclude_range"     "exclude_second"    "filtered_numbers" 
[22] "first_car_name"    "first_row"         "fruits"           
[25] "gender"            "i"                 "is_valid"         
[28] "list_data"         "matrix_data"       "miles_per_hour"   
[31] "more_than_4_cyl"   "mpg"               "mtcars"           
[34] "myString"          "N"                 "name"             
[37] "numbers"           "older_students"    "range_numbers"    
[40] "rep_each"          "rep_times"         "result"           
[43] "selected_numbers"  "sentence"          "seq_1_to_10"      
[46] "seq_by_2"          "seq_length_out"    "some_fruits"      
[49] "students"          "subset_df"         "test_result"      
[52] "third_number"      "vectorized_time"   "weight"           
```

```r
# clear all values
rm(list = ls())
ls()
```

```
character(0)
```

- We can also click on the "Broom" icon in the Environment tab to clear all objects.
- Alternatively, use the `Session > Clear Workspace...` menu option.
- Finally, restarting the R session is a surefire way to clear all objects and start fresh. In RStudio, you can do this via the menu `Session > Restart R` or by using the `Ctrl+Shift+F10` shortcut on Windows/Linux or `Cmd+Shift+F10` on macOS.

Flow of Control

Repetition in R (Loops)

Loops are fundamental constructs in programming that allow you to execute a block of code repeatedly. In R, the primary looping constructs are the `for` loop, the `while` loop, and the `repeat` loop. This section will guide you through the syntax and usage of each, with examples to illustrate how they can be used in data analysis tasks.

The `for` Loop

The `for` loop is used to iterate over a sequence or a vector and execute a block of code for each element.

Syntax:

```r
for (variable in sequence) {
  # Code to execute for each element of the sequence
}
```

Example:

Let's iterate over a vector of numbers and print each number squared.

```r
# Create a vector of numbers
numbers <- 1:5

# Use a for loop to iterate and print squares
for (num in numbers) {
  print(num^2)
}
```

```
[1] 1
[1] 4
[1] 9
```

```
[1] 16
[1] 25
```

What are the correct values for the *s in the code below so that it will iterate over all the items of x array?

```
x <- 50:60
for (i in *:*)
{
  print(x[i])
}
```

In this example, the loop runs five times, squaring each number from 1 to 5.

The `while` Loop

The `while` loop continues to execute as long as a specified condition is true. It's useful when the number of iterations is not known beforehand.

Syntax:

```
while (condition) {
  # Code to execute as long as the condition is true
}
```

Example:

We will use a `while` loop to simulate a countdown.

```
# Set the starting point of the countdown
countdown <- 5

# Begin the while loop
while (countdown > 0) {
  print(paste("Counting down:", countdown))
  countdown <- countdown - 1
}
```

```
[1] "Counting down: 5"
[1] "Counting down: 4"
[1] "Counting down: 3"
[1] "Counting down: 2"
[1] "Counting down: 1"
```

```
# When the loop ends
print("Lift off!")
```

```
[1] "Lift off!"
```

This loop will run until `countdown` is no longer greater than 0.

The `repeat` Loop

The `repeat` loop executes an indefinite number of times until explicitly told to stop with a `break` statement. It's the most basic form of loop and is less commonly used than `for` and `while` loops due to its potential to create infinite loops if not handled carefully.

Syntax:

```
repeat {
  # Code to execute indefinitely
  if (stop_condition) {
    break
  }
}
```

Example:

Here's a `repeat` loop that finds the first number divisible by 13 greater than 100.

```
# Initialize the number
number <- 101

# Start the repeat loop
repeat {
  # Check if the number is divisible by 13
  if (number %% 13 == 0) {
    print(paste("Found a number divisible by 13:", number))
    break
  }
  # Increment the number
  number <- number + 1
}
```

```
[1] "Found a number divisible by 13: 104"
```

The loop runs indefinitely, incrementing `number` until it finds one that satisfies the condition.

Nested Loops

Loops can be nested inside other loops, which is useful for iterating over multiple dimensions, such as rows and columns in a matrix.

Example:

```
# Create a 3x3 matrix
my_matrix <- matrix(1:9, nrow = 3)

# Iterate over each element of the matrix
for (row in 1:nrow(my_matrix)) {
  for (col in 1:ncol(my_matrix)) {
    # Multiply each element by 2
    my_matrix[row, col] <- my_matrix[row, col] * 2
  }
}
print(my_matrix)
```

```
     [,1] [,2] [,3]
[1,]    2    8   14
[2,]    4   10   16
[3,]    6   12   18
```

In this nested loop example, every element of `my_matrix` is doubled.

Conditional Branching in R

Introduction to Conditional Branching

Conditional branching in R allows you to control the flow of execution based on conditions. It's a fundamental concept in programming that lets you execute certain sections of code while skipping others, depending on whether specified conditions are `TRUE` or `FALSE`. This section will explain how to use conditional statements in R, such as `if`, `else`, and `else if`, along with practical examples.

The `if` Statement

The `if` statement is the simplest form of conditional branching. It evaluates a condition and, if that condition is `TRUE`, it executes a block of code.

Syntax:

```
if (condition) {
  # Code to execute if the condition is TRUE
}
```

Example:

Here, we check if a number is positive.

```
number <- 5

if (number > 0) {
  print("The number is positive.")
}
```

```
[1] "The number is positive."
```

The `else` Statement

The `else` statement is used together with an `if` statement to execute code when the `if` condition is not `TRUE`.

Syntax:

```
if (condition) {
  # Code to execute if the condition is TRUE
} else {
  # Code to execute if the condition is FALSE
}
```

Example:

Expanding on the previous example, we also handle the case when the number is not positive.

```
number <- -3

if (number > 0) {
  print("The number is positive.")
} else {
  print("The number is not positive.")
}
```

```
[1] "The number is not positive."
```

The `else if` Statement

The `else if` statement allows you to check multiple conditions. If the condition for `if` is `FALSE`, it checks the condition for `else if`, and so on.

Syntax:

```
if (condition1) {
  # Code to execute if condition1 is TRUE
} else if (condition2) {
  # Code to execute if condition1 is FALSE and condition2 is TRUE
} else {
  # Code to execute if none of the above conditions are TRUE
}
```

Example:

Here, we classify a number as positive, negative, or zero.

```
number <- 0

if (number > 0) {
  print("The number is positive.")
} else if (number < 0) {
  print("The number is negative.")
} else {
  print("The number is zero.")
}
```

```
[1] "The number is zero."
```

The `switch` Statement

The `switch` statement is a multi-way branch used when you want to compare the same variable (or expression) with many different possible values and execute different pieces of code for each value.

Syntax:

```
switch(EXPR,
       "case1" = {# Code for case 1},
       "case2" = {# Code for case 2},
       # ...,
       {# Default case code}
)
```

Example:

The following example shows how `switch` can be used to print different messages based on the value of a character variable.

```
day <- "Tue"

switch(day,
       "Mon" = {print("Start of the work week")},
       "Tue" = {print("Second day of the work week")},
       "Wed" = {print("Middle of the work week")},
       "Thu" = {print("Approaching the end")},
       "Fri" = {print("Last day of the work week")},
       {print("It must be the weekend")}
)
```

```
[1] "Second day of the work week"
```

Creating Functions in R

Introduction

Functions in R are crucial for efficient and organized code, allowing for repetitive tasks to be encapsulated into single, callable entities. This chapter demonstrates how to craft your own functions, enabling the automation of tasks and making your code more modular and readable.

Basic Function Structure

A function is defined using the `function` keyword, followed by parentheses containing any arguments, and a body enclosed in curly braces.

Example: Simple Addition Function

```r
add_two_numbers <- function(a, b) {
  result <- a + b
  return(result)
}

# Using the function
print(add_two_numbers(5, 3))
```

```
[1] 8
```

Function Arguments and Return Values

Functions can take arguments and return results. R functions return the result of the last line in the function body by default, but the `return()` function can be used for clarity or early returns.

Example: Greeting Function

```r
greet_person <- function(name) {
  greeting <- paste("Hello,", name)
  return(greeting)
}

# Using the function
print(greet_person("Alice"))
```

```
[1] "Hello, Alice"
```

Default Argument Values

You can specify default values for arguments, making them optional.

Example: Power Function

```r
raise_to_power <- function(base, exponent = 2) {
  result <- base ^ exponent
  return(result)
}

# Using the function with default exponent
print(raise_to_power(4))
```

```
[1] 16
```

```r
# Specifying both arguments
print(raise_to_power(4, 3))
```

```
[1] 64
```

Returning Multiple Values

Functions in R can return multiple values using a list.

Example: Statistics Function

```r
calculate_stats <- function(numbers) {
  mean_val <- mean(numbers)
  sum_val <- sum(numbers)
  return(list(mean = mean_val, sum = sum_val))
}

# Using the function
stats <- calculate_stats(c(1, 2, 3, 4, 5))
print(stats)
```

```
$mean
[1] 3

$sum
[1] 15
```

Anonymous Functions

For short, one-time operations, anonymous functions can be used.

Example: Doubling Numbers

```r
numbers <- 1:5
doubled_numbers <- sapply(numbers, function(x) x * 2)
print(doubled_numbers)
```

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

Scope and Environment

Variables defined inside a function are local to that function.

Example: Increment Function

```r
increment <- function(x) {
  y <- x + 1
  return(y)
}

incremented_value <- increment(5)
print(incremented_value)
```

```
[1] 6
```

```
        # Trying to print 'y' here would result in an error because 'y' is not defined in this s
```