3. Data Structures

July 20, 2023

The R programming language includes a number of data structures that are frequently employed in data analysis and statistical modeling. These are some of the most popular data structures in R:

1. **Vector**: A vector is a one-dimensional array that stores identical data types, such as numeric, character, or logical. The `c()` function can be used to create vectors, and indexing can be used to access individual vector elements.

2. **Factor**: A factor is a vector representing **categorical** data, with each distinct value or category represented as a level. Using indexing, individual levels of a factor can be accessed using the `factor()` function.

3. **Dataframe**: A data frame is a two-dimensional table-like structure similar to a spreadsheet, that can store various types of data in columns. The `data.frame()` function can be used to construct data frames, and individual elements can be accessed using row and column indexing.

4. **Matrix**: A matrix is a two-dimensional array of data with identical rows and columns. The `matrix()` function can be used to construct matrices, and individual elements can be accessed using row and column indexing.

5. **Array**: An array is a multidimensional data structure that can contain data of the same data type in user-specified dimensions. Arrays can be constructed using the `array()` function, and elements can be accessed using multiple indexing.

6. **List**: A list is an object that may comprise elements of various data types, including vectors, matrices, data frames, and even other lists. The `list()` function can be used to construct lists, while indexing can be used to access individual elements.

These data structures are helpful for storing and manipulating data in R, and they can be utilized in numerous applications, such as statistical analysis and data visualization. We will focus our attention on Vectors, Factors and Dataframes, since we believe that these are the three most useful data structures. [1]

Vectors

1. A vector is a fundamental data structure in R that can hold a sequence of values of the same data type, such as integers, numeric, character, or logical values.

2. A vector can be created using the `c()` function.

3. R supports two forms of vectors: atomic vectors and lists. Atomic vectors are limited to containing elements of a single data type, such as numeric or character. Lists, on the other hand, can contain elements of various data types and structures. [1]

Vectors in R

1. The following R code creates a numeric vector, a character vector and a logical vector respectively.

```
# Read data into vectors
names <- c("Ashok", "Bullu", "Charu", "Divya")
ages <- c(72, 49, 46, 42)
weights <- c(65, 62, 54, 51)
income <- c(-2, 8, 19, 60)
females <- c(FALSE, TRUE, TRUE, TRUE)
```

2. The `c()` function is employed to combine the four character elements into a single vector.
3. Commas separate the elements of the vector within the parentheses.
4. Individual elements of the vector can be accessed via indexing, which utilizes square brackets []. For instance, `names\[1\]` returns `Ashok`, while `names\[3\]` returns `Charu`.
5. We can also perform operations such as categorizing and filtering on the entire vector. For instance, `sort(names)` returns a vector of sorted names, whereas names[names!= "Bullu"] returns a vector of names excluding `Bullu`.

Vector Operations

Vectors can be used to perform the following vector operations:

1. **Accessing Elements:** We can use indexing with square brackets to access individual elements of a vector. To access the second element of the `names` vector, for instance, we can use:

```
names[2]
```

```
[1] "Bullu"
```

This returns `Bullu`, the second element of the `people` vector.

2. **Concatenation:** The `c()` function can be used to combine multiple vectors into a single vector. For instance, to combine the `names` and `ages` vectors into the "people" vector, we can use:

```
persons <- c(names, ages)
persons
```

```
[1] "Ashok" "Bullu" "Charu" "Divya" "72"    "49"    "46"    "42"
```

This generates an eight-element vector containing the names and ages of the four people.

3. **Subsetting:** We can use indexing with a logical condition to construct a new vector that contains a subset of elements from an existing vector. For instance, to construct a new vector named `female_names` containing only the female names, we can use:

```
female_names <- names[females == TRUE]
female_names
```

```
[1] "Bullu" "Charu" "Divya"
```

This generates a new vector comprising three elements containing the names of the three females `Bullu`, `Charu`, and `Divya`.

4. **Arithmetic Operations:** We can perform element-wise arithmetic operations on vectors.

```
# Addition
addition <- ages + weights
print(addition)
```

```
[1] 137 111 100  93
```

```
# Subtraction
subtraction <- ages - weights
```

```
print(subtraction)
```

```
[1]    7 -13  -8  -9
```

```
# Multiplication
multiplication <- ages * weights
print(multiplication)
```

```
[1] 4680 3038 2484 2142
```

```
# Division
division <- ages / weights
print(division)
```

```
[1] 1.1076923 0.7903226 0.8518519 0.8235294
```

```
# Exponentiation
exponentiation <- ages^2
print(exponentiation)
```

```
[1] 5184 2401 2116 1764
```

In the above code, we perform addition, subtraction, multiplication, division, and exponentiation on these vectors using the arithmetic operators +, -, *, /, and ^ respectively.

In addition to the common arithmetic operations (addition, subtraction, multiplication, division, and exponentiation), R also supports other arithmetic operations such as modulus, integer division, and absolute value. Let's demonstrate these operations

```
# Modulus
modulus <- ages %% income
print(modulus)
```

```
[1]  0  1  8 42
```

```r
# Integer Division
integer_division <- ages %/% income
print(integer_division)
```

```
[1] -36   6   2   0
```

```r
# Absolute Value
absolute_value <- abs(ages)
print(absolute_value)
```

```
[1] 72 49 46 42
```

Let's explore a few additional arithmetic operations:

```r
# Floor Division
floor_division <- floor(ages / income)
print(floor_division)
```

```
[1] -36   6   2   0
```

```r
# Ceiling Division
ceiling_division <- ceiling(ages / income)
print(ceiling_division)
```

```
[1] -36   7   3   1
```

```r
# Logarithm
logarithm <- log(ages)
print(logarithm)
```

```
[1]  4.276666 3.891820 3.828641 3.737670
```

```
# Square Root
square_root <- sqrt(ages)
print(square_root)
```

[1] 8.485281 7.000000 6.782330 6.480741

```
# Sum
sum_total <- sum(ages)
print(sum_total)
```

[1] 209

`floor` calculates the largest integer not exceeding the quotient.

`ceiling` calculates the smallest integer not less than the quotient.

`log` calculates the natural logarithm of each element.

`sum` calculates the sum of all the elements.

5. **Logical Operations:** We can perform logical operations on vectors, which are also executed element-by-element.

```
# Equality comparison
age_equal_46 <- (ages == 46)
print(age_equal_46)
```

[1] FALSE FALSE TRUE FALSE

```
# Inequality comparison
weight_not_equal_54 <- (weights != 54)
print(weight_not_equal_54)
```

[1] TRUE TRUE FALSE TRUE

```
# Logical AND
female_and_income <- females & (income > 0)
```

6

```
print(female_and_income)
```

[1] FALSE   TRUE   TRUE   TRUE

```
# Logical OR
age_or_weight_greater_50 <- (ages > 50) | (weights > 50)
print(age_or_weight_greater_50)
```

[1] TRUE TRUE TRUE TRUE

```
# Logical NOT
not_female <- !females
print(not_female)
```

[1]   TRUE FALSE FALSE FALSE

In the above code, we perform the following logical operations:

Equality Comparison (==): It checks if the elements of the ages vector are equal to 46. The resulting vector, age_equal_46, contains TRUE for elements that are equal to 46 and FALSE otherwise.

Inequality Comparison (!=): It checks if the elements of the weights vector are not equal to 54. The resulting vector, weight_not_equal_54, contains TRUE for elements that are not equal to 54 and FALSE otherwise.

Logical AND (&): It performs a logical AND operation between the females vector and the condition (income > 0). The resulting vector, female_and_income, contains TRUE for elements that satisfy both conditions and FALSE otherwise.

Logical OR (|): It performs a logical OR operation between the conditions (ages > 50) and (weights > 50). The resulting vector, age_or_weight_greater_50, contains TRUE for elements that satisfy either condition or both.

Logical NOT (!): It negates the values in the females vector. The resulting vector, not_female, contains TRUE for elements that were originally FALSE and FALSE for elements that were originally TRUE.

```r
# Greater than comparison
age_greater_50 <- ages > 50
print(age_greater_50)
```

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

```r
# Less than or equal to comparison
weight_less_equal_54 <- weights <= 54
print(weight_less_equal_54)
```

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

```r
# Element-wise AND
age_and_weight_greater_50 <- (ages > 50) & (weights > 50)
print(age_and_weight_greater_50)
```

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

```r
# Element-wise OR
age_or_weight_less_equal_50 <- (ages > 50) | (weights <= 50)
print(age_or_weight_less_equal_50)
```

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

```r
# Element-wise XOR
age_xor_weight_greater_50 <- xor(ages > 50, weights > 50)
print(age_xor_weight_greater_50)
```

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

```r
# Any True
any_female <- any(females)
print(any_female)
```

```
[1] TRUE
```

```
# All True
all_female <- all(females)
print(all_female)
```

[1] FALSE

Greater than Comparison (>): It checks if each element of the ages vector is greater than 50. The resulting vector, age_greater_50, contains TRUE for elements that satisfy the condition and FALSE otherwise.

Less than or Equal to Comparison (<=): It checks if each element of the weights vector is less than or equal to 54. The resulting vector, weight_less_equal_54, contains TRUE for elements that satisfy the condition and FALSE otherwise.

Element-wise AND (&): It performs an element-wise logical AND operation between the conditions (ages > 50) and (weights > 50). The resulting vector, age_and_weight_greater_50, contains TRUE for elements that satisfy both conditions and FALSE otherwise.

Element-wise OR (|): It performs an element-wise logical OR operation between the conditions (ages > 50) and (weights <= 50). The resulting vector, age_or_weight_less_equal_50, contains TRUE for elements that satisfy either condition or both.

Element-wise XOR (xor()): It performs an element-wise exclusive OR operation between the conditions (ages > 50) and (weights > 50). The resulting vector, age_xor_weight_greater_50, contains TRUE for elements where exactly one condition is true and FALSE otherwise.

Any True (any()): It checks if at least one element in the females vector is TRUE. The result, any_female, is TRUE if there is at least one TRUE value in the vector and FALSE otherwise.

All True (all()): It checks if all elements in the females vector are TRUE. The result, all_female, is TRUE if all values in the vector are TRUE and FALSE otherwise.

```
# Negation
not_female <- !females
print(not_female)
```

[1] TRUE FALSE FALSE FALSE

```
# Any True
any_age_greater_50 <- any(ages > 50)
print(any_age_greater_50)
```

```
[1] TRUE
```

```r
# All True
all_income_positive <- all(income > 0)
print(all_income_positive)
```

```
[1] FALSE
```

```r
# Subset with Logical Vector
female_names <- names[females]
print(female_names)
```

```
[1] "Bullu" "Charu" "Divya"
```

```r
# Combined Logical Operation
combined_condition <- (ages > 50 & weights <= 54) | (income > 0 & females)
print(combined_condition)
```

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

```r
# Logical Function anyNA()
has_na <- anyNA(names)
print(has_na)
```

```
[1] FALSE
```

```r
# Logical Function is.na()
is_na <- is.na(ages)
print(is_na)
```

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

```
# Finding unique values
unique(ages)
```

```
[1] 72 49 46 42
```

Negation (!): It negates the values in the females vector. The resulting vector, not_female, contains TRUE for elements that were originally FALSE and FALSE for elements that were originally TRUE.

Any True (any()): It checks if there is at least one TRUE value in the logical vector ages > 50. The result, any_age_greater_50, is TRUE if at least one element in ages is greater than 50 and FALSE otherwise.

All True (all()): It checks if all elements in the logical vector income > 0 are TRUE. The result, all_income_positive, is TRUE if all values in the income vector are greater than 0 and FALSE otherwise.

Subset with Logical Vector: It uses a logical vector females to subset the names vector. The resulting vector, female_names, contains only the names where the corresponding element in females is TRUE.

Combined Logical Operation: It combines multiple conditions using logical AND (&) and logical OR (|). The resulting vector, combined_condition, contains TRUE for elements that satisfy the combined condition and FALSE otherwise.

Logical Function anyNA(): It checks if there are any missing values (NA) in the names vector. The result, has_na, is TRUE if there is at least one NA value and FALSE otherwise.

Logical Function is.na(): It checks if each element of the ages vector is NA. The resulting vector, is_na, contains TRUE for elements that are NA and FALSE otherwise.

unique(): It finds the unique values in the `ages` vector

6. **Sorting:** We can sort a vector in ascending or descending order using the `sort()` function. For example, to sort the `ages` vector in descending order, we can use:

```
# Sort in ascending order
sorted_names <- sort(names)
print(sorted_names)
```

```
[1] "Ashok" "Bullu" "Charu" "Divya"
```

```r
# Sort in descending order
sorted_names_desc <- sort(names, decreasing = TRUE)
print(sorted_names_desc)
```

```
[1] "Divya" "Charu" "Bullu" "Ashok"
```

In the above code, we demonstrate sorting the names vector in both ascending and descending order using the sort() function. By default, sort() sorts the vector in ascending order. To sort in descending order, we set the decreasing argument to TRUE.

Statistical Operations on Vectors

1. **Length**: The length represents the count of the number of elements in a vector.

```r
length(ages)
```

```
[1] 4
```

2. **Maximum** and **Minimum**: The maximum and minimum values are the vector's greatest and smallest values, respectively.

3. **Range**: The range is a measure of the spread that represents the difference between the maximum and minimum values in a vector.

```r
min(ages)
```

```
[1] 42
```

```r
max(ages)
```

```
[1] 72
```

```r
range(ages)
```

```
[1] 42 72
```

4. **Mean**: The mean is a central tendency measure that represents the average value of a vector's elements.

5. **Standard Deviation**: The standard deviation is a measure of dispersion that reflects the amount of variation in a vector's elements.

```
mean(ages)
```

```
[1] 52.25
```

```
sd(ages)
```

```
[1] 13.47529
```

6. **Median**: The median is a measure of central tendency that represents the middle value of a sorted vector.

```
median(ages)
```

```
[1] 47.5
```

7. **Quantiles**: The quantiles are a set of cut-off points that divide a sorted vector into equal-sized groups.

```
quantile(ages)
```

```
   0%    25%    50%    75%   100%
42.00  45.00  47.50  54.75  72.00
```

This will return a set of five values, representing the minimum, first quartile, median, third quartile, and maximum of the four ages.

8. Additional Functionality:

```
# Standard Error of the Mean
se_ages <- sqrt(var(ages) / length(ages))
print(se_ages)
```

```
[1] 6.737643
```

```
# Cumulative Sum
cumulative_sum_ages <- cumsum(ages)
print(cumulative_sum_ages)
```

[1] 72 121 167 209

```
# Correlation Coefficient
correlation_ages_females <- cor(ages, females)
print(correlation_ages_females)
```

[1] -0.9770974

Standard Error of the Mean: It calculates the standard error of the mean for the ages vector. The result is stored in se_ages.

Cumulative Sum: It calculates the cumulative sum of the elements in the ages vector. The cumulative sum is stored in cumulative_sum_ages.

Correlation Coefficient: It calculates the correlation coefficient between the ages and females vectors using the cor() function. T

Thus, we note that the R programming language provides a wide range of statistical operations that can be performed on vectors for data analysis and modeling. Vectors are clearly a potent and versatile data structure that can be utilized in a variety of ways.

Strings

Here are some common string operations that can be conducted using the provided vector examples.

1. **Substring**: The substr() function can be used to extract a substring from a character vector. To extract the first three characters of each name in the "names" vector, for instance, we can use:

```
substring_names <- substr(names, start = 2, stop = 4)
print(substring_names)
```

[1] "sho" "ull" "har" "ivy"

This returns a new character vector containing three letters of each name.

2. **Concatenation**: Using the `paste()` function, we can concatenate two or more character vectors into a singular vector. To create a new vector containing the names and ages of the individuals, for instance, we can use:

```r
persons <- paste(names, ages)
print(persons)
```

```
[1] "Ashok 72" "Bullu 49" "Charu 46" "Divya 42"
```

```r
full_names <- paste(names, "Kumar")
print(full_names)
```

```
[1] "Ashok Kumar" "Bullu Kumar" "Charu Kumar" "Divya Kumar"
```

This will generate a new eight-element character vector containing the name and age of each individual, separated by a space.

3. **Case Conversion:** The `toupper()` and `tolower()` functions can be used to convert the case of characters within a character vector. To convert the "names" vector to uppercase letters, for instance, we can use:

```r
toupper(names)
```

```
[1] "ASHOK" "BULLU" "CHARU" "DIVYA"
```

This will generate a new character vector with all of the names converted to uppercase.

4. **Pattern Matching:** Using the `grep()` and `grepl()` functions, we can search for a pattern within the elements of a character vector. To find the names in the "names" vector that contain the letter "a", for instance, we can use:

```r
grep("a", names)
```

```
[1] 3 4
```

This returns a vector containing the indexes of the "names" vector elements that contain the letter "a."

```r
pattern_match <- grep("l", names, value = TRUE)
print(pattern_match)
```

```
[1] "Bullu"
```

```r
# Length of Strings
name_lengths <- nchar(names)
print(name_lengths)
```

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

```r
# %in% Operator
names_found <- names %in% c("Ashok", "Charu")
print(names_found)
```

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

```r
# Logical Function ifelse()
age_category <- ifelse(ages > 50, "Old", "Young")
print(age_category)
```

```
[1] "Old"   "Young" "Young" "Young"
```

%in% Operator: It checks if each element in the names vector is present in the specified set of names. The resulting vector, names_found, contains TRUE for elements that are found in the set and FALSE otherwise.

Logical Function ifelse(): It evaluates a logical condition and returns values based on the condition. In this example, we use ifelse() to assign the value "Old" to elements in the age_category vector where the corresponding element in ages is greater than 50, and "Young" otherwise.

References

[1] R Core Team. (2021). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. **https://www.R-project.org/**

R Core Team. (2022). Vectors, Lists, and Arrays. R Documentation. https://cran.r-project.org/doc/manuals/r-release/R-intro.html#vectors-lists-and-arrays

Wickham, H., & Grolemund, G. (2016). R for data science: Import, tidy, transform, visualize, and model data. O'Reilly Media, Inc.