Introduction to R

POL-501

2024-08-27

Table of Contents

# 1 R Basics

## 1.1 Using R as a Calculator

At its simplest, you can use R as a basic calculator.

#You can perform addition...  
3 + 5 # Displays 8

## [1] 8

#...subtraction...  
10 - 7 # Displays 3

## [1] 3

#...multiplication...  
4 \* 2 # Displays 8

## [1] 8

#...and division.  
10 / 5 # Displays 2

## [1] 2

## 1.2 Object Assignment

R allows us to assign values to objects (which can be thought of as variables). To assign a value to an object in R, we use the “<-” symbol.

# Example of object assignment:  
x <- 10 # Assigning the value 10 to the object 'x'.  
print(x) # Printing the value of 'x'. This will output 10.

## [1] 10

## 1.3 Data Types in R

In R, we have several types of data. These include:

1. Numeric: Any number with or without decimal points.
2. Integer: Whole numbers without decimal points.
3. Complex: Complex numbers, comprising a real and imaginary part.
4. Character: Text or string data.
5. Logical: Boolean values - TRUE or FALSE.

Let’s assign each of these types to a variable:

# Creating a numeric variable  
numeric\_var <- 5.3  
print(numeric\_var) # This will output 5.3.

## [1] 5.3

# Creating an integer variable  
integer\_var <- 4L  
print(integer\_var) # This will output 4.

## [1] 4

# Creating a complex variable  
complex\_var <- 3 + 2i  
print(complex\_var) # This will output 3+2i.

## [1] 3+2i

# Creating a character variable  
character\_var <- "Hello, World!"  
print(character\_var) # This will output "Hello, World!".

## [1] "Hello, World!"

# Creating a logical variable  
logical\_var <- TRUE  
print(logical\_var) # This will output TRUE.

## [1] TRUE

## 1.4 Vectors/List in R

One of the key data structures in R is a vector. Vectors are one-dimensional arrays that can store numeric, character or logical data elements.

# Here’s an example of a numeric vector:  
numeric\_vector <- c(1, 2, 3)  
print(numeric\_vector) # This will output "1 2 3".

## [1] 1 2 3

# Here’s an example of a character vector:  
character\_vector <- c("Welcome", "to", "R!")  
print(character\_vector) # This will output "Welcome to R!".

## [1] "Welcome" "to" "R!"

# Here's an example of a logical vector  
logical\_vector <- c(TRUE, FALSE, TRUE, FALSE)  
print(logical\_vector) # This will output "TRUE FALSE TRUE FALSE".

## [1] TRUE FALSE TRUE FALSE

## 1.5 Boolean in R

R has logical operators to perform operations involving booleans (TRUE and FALSE).

# Here's an example of using boolean values in R:  
x <- 5  
y <- 10  
  
# Check if x is less than y  
result <- x < y # This will return TRUE since 5 is indeed less than 10.  
  
print(result)

## [1] TRUE

# 2 Handling Dataframes: An Introductory Example.

This detailed guide instructs about fundamental R operations through a realistic example — a classroom scenario. It will cover creating data frames, alteration of data frames, plotting, working with categorical data (factors), and handling missing data in R.

Let’s imagine we are teachers gathering students’ data such as name, age, scores in recent tests, and gender.

## 2.1 Building Data Frames

Data frames in R function similar to tables in a database or Excel spreadsheet, with rows and columns that store data.

# Constructing a data frame named 'studentdata'.  
# Our data includes "Name", "Age", "Gender", and "Score" of each student.  
  
studentdata <- data.frame(  
 Name = c("John", "Alice", "Bob", "Emily", "Jack", "Sandy"),  
 Age = c(23, 35, 45, 22, 33, 29),  
 Gender = c("Male", "Female", "Male", "Non-binary", "Male", "Female"),  
 Score = c(89, 92, 76, 88, 75, 95)  
)  
  
print(studentdata)

## Name Age Gender Score  
## 1 John 23 Male 89  
## 2 Alice 35 Female 92  
## 3 Bob 45 Male 76  
## 4 Emily 22 Non-binary 88  
## 5 Jack 33 Male 75  
## 6 Sandy 29 Female 95

## 2.2 Modifying Data Frames

Adding and removing columns to a data frame are routine tasks - requirement changes might need adding new variables or erasing irrelevant ones.

# Adding a new column  
# A new column "Passed" is added indicating if each student passed or not (a score above 80 is deemed a pass)  
  
studentdata$Passed <- studentdata$Score > 80  
  
# Removing a column  
# Now, we opt to remove the 'Score' column as it's not required after we ascertain who's passed.  
  
studentdata$Score <- NULL  
  
print(studentdata)

## Name Age Gender Passed  
## 1 John 23 Male TRUE  
## 2 Alice 35 Female TRUE  
## 3 Bob 45 Male FALSE  
## 4 Emily 22 Non-binary TRUE  
## 5 Jack 33 Male FALSE  
## 6 Sandy 29 Female TRUE

Accessing parts of a data frame is similar to picking specific cells or a range in a spreadsheet, let’s see how:

# Accessing a specific element - 'Name' of the first student.  
print(studentdata[1, "Name"])

## [1] "John"

# Accessing an entire column - 'Age' of all students  
print(studentdata$Age)

## [1] 23 35 45 22 33 29

## 2.3 Factors

Factors handle categorical data in R - understanding this is essential to handle and analyze categorical variables.

# We've already stored the gender of each student in the 'Gender' column.  
# Let's convert that into factor.  
studentdata$Gender <- factor(studentdata$Gender)  
  
# Checking levels in Gender  
print(levels(studentdata$Gender))

## [1] "Female" "Male" "Non-binary"

## 2.4 Handling Missing Data

Real-world datasets often contain missing values. Handling missing data can involve techniques such as detection, deletion, and imputation.

# Let's presume a scenario where student Age data has missing values.  
# 'NA' denotes missing value in R.  
AgeData <- c(23, 35, NA, 22, NA, 29)  
  
# Identifying missing values with 'is.na'  
print(is.na(AgeData))

## [1] FALSE FALSE TRUE FALSE TRUE FALSE

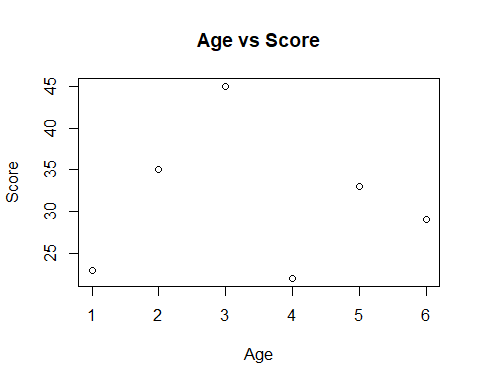
# Replacing missing values via a specific value, let's replace 'NA' with median age of students.  
Age\_median <- median(AgeData, na.rm = TRUE)  
AgeData[is.na(AgeData)] <- Age\_median  
  
print(AgeData)

## [1] 23 35 26 22 26 29

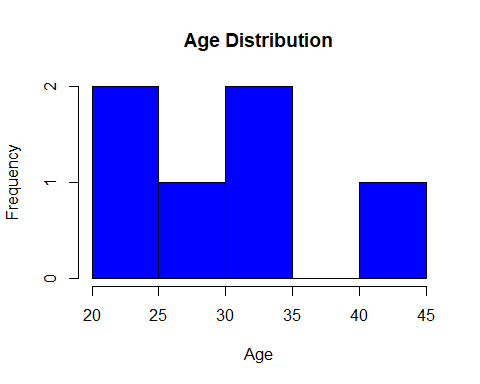
## 2.5 Plotting Data

Visualization broadens the understanding of data by highlighting patterns and distributions in a visually intuitive manner.

# A scatter plot between Age and Score of students.  
plot(studentdata$Age, studentdata$Score, main = "Age vs Score", xlab = "Age", ylab = "Score")



# A histogram represents 'Age' distribution among the students.  
hist(studentdata$Age, main = "Age Distribution", xlab = "Age", col = "blue")



# 3 USArrests Dataset

Let’s now use this knowledge to explore the USArrests dataset. Please note that to use the USArrests dataset, it requires the ‘datasets’ package to be installed and loaded. The USArrests dataset contains statistics about violent crimes in US states.

Variables Included:

* Murder: Murder arrests (per 100,000)
* Assault: Assault arrests (per 100,000)
* UrbanPop: Percent urban population
* Rape: Rape arrests (per 100,000)

# Install 'datasets'  
install.packages('datasets') # You just need to install packages once. Then you can comment this line with '#'

## Warning: package 'datasets' is in use and will not be installed

# Loading the USArrests data  
data(USArrests)  
  
# Printing the first few entries of the USArrests dataset  
print(head(USArrests))

## Murder Assault UrbanPop Rape  
## Alabama 13.2 236 58 21.2  
## Alaska 10.0 263 48 44.5  
## Arizona 8.1 294 80 31.0  
## Arkansas 8.8 190 50 19.5  
## California 9.0 276 91 40.6  
## Colorado 7.9 204 78 38.7

## 3.1 Subsetting in R

Let’s first look at the structure of the USArrests dataset:

# The function str() provides a neat, human-readable summary of a complex R data structure  
str(USArrests)

## 'data.frame': 50 obs. of 4 variables:  
## $ Murder : num 13.2 10 8.1 8.8 9 7.9 3.3 5.9 15.4 17.4 ...  
## $ Assault : int 236 263 294 190 276 204 110 238 335 211 ...  
## $ UrbanPop: int 58 48 80 50 91 78 77 72 80 60 ...  
## $ Rape : num 21.2 44.5 31 19.5 40.6 38.7 11.1 15.8 31.9 25.8 ...

This output provides us crucial information about our dataset, such as there are 50 observations (rows) and 4 variables (columns).

Subsetting in R is a way to extract parts of your data from larger data structures based on certain conditions, generally done using the square bracket notation [row, col]. The concept works on principles of rows and columns, similar to how we understand data in a spreadsheet. Here, ‘row’ and ‘col’ represent the indices of the rows and columns respectively.

For instance, if we have a dataset ‘USArrests’, we can extract data from the second row and third column by specifying the indices as USArrests[2, 3]. In more realistic scenarios, you will want to subset data based on certain conditions, which is where logical subsetting is useful.

### 3.1.1 Basic Subsetting

Subsetting is accomplished primarily through the use of square brackets [].

# To select the 'Murder' column from the USArrests dataset, we can use  
murder\_data <- USArrests[c("Murder")]  
  
# This line of code gives us the 'Murder' rate for the different states. Let's print the first few values.  
print(head(murder\_data))

## Murder  
## Alabama 13.2  
## Alaska 10.0  
## Arizona 8.1  
## Arkansas 8.8  
## California 9.0  
## Colorado 7.9

### 3.1.2 Quick Access Subsetting

If we only need a single entry from our dataset. For instance, let’s say we want to know the Assault rate of the tenth state in the dataset. We can subset this as follows:

# Assault rate for the 10th state  
tenth\_state\_assault <- USArrests[10, "Assault"]  
  
# Printing the Assault rate of the 10th state  
print(tenth\_state\_assault)

## [1] 211

### 3.1.3 Condition-Based Subsetting

Quite often, we want to subset our data based on certain conditions. To achieve this, we can use logical conditions inside our square brackets.

The condition needs to generate a list of boolean (TRUE/FALSE) values of equal length to the number of rows in the data structure. This list tells R which rows to keep (for TRUE) and which to exclude (for FALSE).

We can create these logical conditions manually or use a logical statement. For example, we might want to subset rows of the dataset where the ‘Murder’ rate is above 5. We can create a logical condition ‘logical\_condition <- USArrests[,“Murder”] > 5’, which returns a list of TRUE or FALSE values for each row, depending on whether that row’s ‘Murder’ rate is higher than 5. Subsequently, we subset the dataframe using this logical\_condition: ‘high\_murder\_rate <- USArrests[logical\_condition ,]’.

This entire process can be concisely written in one line as ‘high\_murder\_rate <- USArrests[USArrests[,“Murder”] > 5,]’.

# Suppose we want only the entries of states that have a murder rate more than 5 per 100,000. We can achieve this by applying a condition on the 'Murder' column.  
  
# First, start by creating a vector with the logical condition.   
logical\_condition <- USArrests[,"Murder"] > 5

This vector has the same number of rows than our dataset, and has the value TRUE whenever , and FALSE otherwise.

Then, we can select the rows of the dataset by subsetting on the logical values of this vector as follows:

# Apply the desired subsetting  
high\_murder\_rate <- USArrests[logical\_condition,]   
  
# Printing the first few rows of the states with high murder rate data.  
print(head(high\_murder\_rate))

## Murder Assault UrbanPop Rape  
## Alabama 13.2 236 58 21.2  
## Alaska 10.0 263 48 44.5  
## Arizona 8.1 294 80 31.0  
## Arkansas 8.8 190 50 19.5  
## California 9.0 276 91 40.6  
## Colorado 7.9 204 78 38.7

We can do the two steps, in one compact line of code in which we obtain ‘high\_murder\_rate’:

# Suppose we want only the entries of states that have a murder rate more than 5 per 100,000. We can achieve this by applying a condition on the 'Murder' column.  
high\_murder\_rate <- USArrests[USArrests[,"Murder"] > 5,]   
  
# Printing the first few rows of the states with high murder rate data.  
print(head(high\_murder\_rate))

## Murder Assault UrbanPop Rape  
## Alabama 13.2 236 58 21.2  
## Alaska 10.0 263 48 44.5  
## Arizona 8.1 294 80 31.0  
## Arkansas 8.8 190 50 19.5  
## California 9.0 276 91 40.6  
## Colorado 7.9 204 78 38.7

This gives us a dataset that only includes states that have a murder rate exceeding our specified threshold.

### 3.1.4 Complex Subsetting

When dealing with diverse datasets like USArrests, we often require subsets based on multiple conditions. R allows for complex subsetting using & (and) or | (or) operators.

# Suppose we want data of states that not only have a high murder rate, but also have an Assault rate more than 150.   
  
high\_murder\_and\_assault <- USArrests[USArrests[,"Murder"] > 5 & USArrests[,"Assault"] > 150,]  
  
# Let's print this data:  
print(high\_murder\_and\_assault) # States matching the criteria

## Murder Assault UrbanPop Rape  
## Alabama 13.2 236 58 21.2  
## Alaska 10.0 263 48 44.5  
## Arizona 8.1 294 80 31.0  
## Arkansas 8.8 190 50 19.5  
## California 9.0 276 91 40.6  
## Colorado 7.9 204 78 38.7  
## Delaware 5.9 238 72 15.8  
## Florida 15.4 335 80 31.9  
## Georgia 17.4 211 60 25.8  
## Illinois 10.4 249 83 24.0  
## Louisiana 15.4 249 66 22.2  
## Maryland 11.3 300 67 27.8  
## Michigan 12.1 255 74 35.1  
## Mississippi 16.1 259 44 17.1  
## Missouri 9.0 178 70 28.2  
## Nevada 12.2 252 81 46.0  
## New Jersey 7.4 159 89 18.8  
## New Mexico 11.4 285 70 32.1  
## New York 11.1 254 86 26.1  
## North Carolina 13.0 337 45 16.1  
## Oklahoma 6.6 151 68 20.0  
## South Carolina 14.4 279 48 22.5  
## Tennessee 13.2 188 59 26.9  
## Texas 12.7 201 80 25.5  
## Virginia 8.5 156 63 20.7  
## Wyoming 6.8 161 60 15.6

We successfully derived a subset of data that matches our criteria.

## 3.2 Changing the name of rows

The following command show the rownames of the dataset. There are different ways that you can name them, but often datasets will have numeric values are row names

# Print Row Names  
rownames(USArrests)

## [1] "Alabama" "Alaska" "Arizona" "Arkansas"   
## [5] "California" "Colorado" "Connecticut" "Delaware"   
## [9] "Florida" "Georgia" "Hawaii" "Idaho"   
## [13] "Illinois" "Indiana" "Iowa" "Kansas"   
## [17] "Kentucky" "Louisiana" "Maine" "Maryland"   
## [21] "Massachusetts" "Michigan" "Minnesota" "Mississippi"   
## [25] "Missouri" "Montana" "Nebraska" "Nevada"   
## [29] "New Hampshire" "New Jersey" "New Mexico" "New York"   
## [33] "North Carolina" "North Dakota" "Ohio" "Oklahoma"   
## [37] "Oregon" "Pennsylvania" "Rhode Island" "South Carolina"  
## [41] "South Dakota" "Tennessee" "Texas" "Utah"   
## [45] "Vermont" "Virginia" "Washington" "West Virginia"   
## [49] "Wisconsin" "Wyoming"

For our purposes, it is useful to have the rownames in a variable inside the dataset. We can create a new variable to do this using the following syntax:

# Print Row Names  
USArrests$State <- rownames(USArrests)  
  
# Get the first values of the variable  
head(USArrests$State)

## [1] "Alabama" "Alaska" "Arizona" "Arkansas" "California"  
## [6] "Colorado"

In R, the $ sign is a special operator used for accessing and manipulating columns within a data frame or list. When working with data frames, $ allows you to reference a specific column by its name and is commonly used to extract or assign values to that column. For example, in the command USArrests$State <- rownames(USArrests), the $ sign is used to create a new column named State in the USArrests data frame and assign the row names of the data frame to this column.

### 3.2.1 How $ Sign Works in Data Frames

When you use the $ sign with a data frame, it follows the syntax:

# Code to create a new variable   
data\_frame$column\_name <- "(some valid expression)"

* Accessing Values: If the column already exists, this syntax allows you to access the values in that column. For instance, USArrests$Murder would give you the values of the Murder column in the USArrests data frame.
* Creating or Modifying a Column: If the column doesn’t exist, using the $ sign with an assignment (<-) will create a new column with that name and fill it with the values on the right-hand side of the assignment. In the example USArrests$State <- rownames(USArrests), a new column State is created, and the row names of USArrests are stored in it. Conditions for Data Frame to Accept New Values

For a data frame to accept new values in a column using the $ operator, a few conditions must be met:

* Matching Row Count: The number of values you assign to the new column must match the number of rows in the data frame. For example, if USArrests has 50 rows, the vector or list assigned to USArrests$State must also contain 50 elements. Otherwise, R will throw an error or recycle the values, which may lead to unexpected results.
* Column Name Validity: The name you provide after the $ sign must be a valid R identifier. This means it should start with a letter and can only contain letters, numbers, periods, or underscores. If you try to create a column name that doesn’t follow these rules, R will either give an error or automatically adjust the name (e.g., replacing spaces with periods).
* No Conflicts with Existing Columns: If the column name already exists in the data frame, the new values will overwrite the existing values in that column. It’s important to be cautious when doing this to avoid unintentionally losing data

# 4 Data Manipulation in R

R provides a comprehensive set of tools for manipulating, transforming, and summarizing data. In this section of the tutorial we will create a new column in the USArrests dataset, which will be a unified crime score. This score will be a single measure that combines the ‘Murder’, ‘Assault’, and ‘Rape’ variables. Following this, we will create a histogram of the crime scores and will summarize this score.

## 4.1 Creating a Unified Crime Score

First, we need to create a new column that combines the ‘Murder’, ‘Assault’, and ‘Rape’ columns. One simple way to do this is to add up the three columns. However, to ensure that each type of crime gets an equal weight in the combined score, we first need to scale the data. We will do this by subtracting the mean of each column and dividing by the standard deviation. This process, also known as standardizing or z-score normalization, gives each column a mean of 0 and a standard deviation of 1.

# Adding packages  
library(dplyr) # the dplyr package gives us access to the mutate(), mean() and sd() functions which will be handy.

##   
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':  
##   
## filter, lag

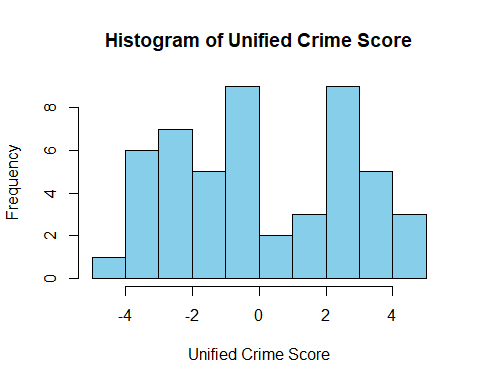
## The following objects are masked from 'package:base':  
##   
## intersect, setdiff, setequal, union

# We standardize the 'Murder', 'Assault', and 'Rape' columns  
USArrests <- USArrests %>%  
 mutate(  
 Murder = (Murder - mean(Murder)) / sd(Murder),  
 Assault = (Assault - mean(Assault)) / sd(Assault),  
 Rape = (Rape - mean(Rape)) / sd(Rape)  
 )  
  
# We create a new column named 'crime\_score' that is the sum of the standardizes 'Murder', 'Assault' and 'Rape' columns  
USArrests <- USArrests %>%  
 mutate(  
 crime\_score = Murder + Assault + Rape  
 )

## 4.2 Creating a Histogram of the Crime Score

Now that we have a unified crime score for each state, we can create a histogram to visualize the distribution of crime scores across all states.

# Generate a histogram of the unified crime score  
hist(USArrests$crime\_score, main="Histogram of Unified Crime Score", xlab="Unified Crime Score", col="skyblue")



## 4.3 Getting Summary Statistics for the Crime Score

Finally, we will generate some summary statistics for the crime score.

# Compute summary statistics of the unified crime score  
summary\_stats <- summary(USArrests$crime\_score)  
print(summary\_stats)

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## -4.6009 -2.1666 -0.5656 0.0000 2.4669 4.8574

The summary() function provides us with the minimum, maximum, mean, and quartile values for the crime score.

Now we have examined the USArrests dataset using various statistical methods including data subsetting, renaming variables, scaling data, creating histograms and getting summary statistics.