**Variables in R Programming**

A variable is a name given to a memory location, which is used to store values in a computer program. Variables in R programming can be used to store numbers (real and complex), words, matrices, and even tables. R is a dynamically programmed language which means that unlike other programming languages, we do not have to declare the data type of a variable before we can use it in our program.  
**For a variable to be valid, it should follow these rules**

* It should contain letters, numbers, and only dot or underscore characters.
* It should not start with a number (eg:- 2iota)
* It should not start with a dot followed by a number (eg:- .2iota)
* It should not start with an underscore (eg:- \_iota)
* It should not be a reserved keyword

### ****Reserved Keywords in R****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Following are the reserved keywords in R | | | | |
| for | in | repeat | while | function |
| if | else | next | break | TRUE |
| FALSE | NULL | Inf | NaN | NA |
| NA\_integer\_ | NA\_real\_ | NA\_complex\_ | NA\_character\_ | … |

**NA:**– Not Available is used to represent missing values.  
**NULL:-**It represents a missing or an undefined value.  
**NaN:**– It is a short form for Not a Number(eg:- 0/0).  
**TRUE/FALSE:** – These are used to represent Logical values.  
**Inf :**– It denotes Infinity(eg:- 1/0).  
**If else, repeat, while, function, for, in, next, and break:–**These are Used as looping statements, conditional statements, and functions.  
…  :- It is used to pass argument settings from one function to another.  
**NA\_integer\_, NA\_real\_, NA\_complex\_, and  NA\_ character \_:-**These represent missing values of other atomic types.  
For example:

* x = 15 implicitly assigns a numeric data type to the variable ‘x’.
* mystring = “Hello, World!”

### ****Constants in R****

The entities whose values are fixed are called constants.  
There are two types of constants in R:

* Numeric Constants: All numeric values such as integer, double, or complex fall under this category. Numeric constants followed by ‘L’ and ‘i’ are considered as integer and complex respectively. And, numeric constants preceded by 0x/0X are treated as hexadecimal numbers.
* Character Constants: These constants are represented by single (‘) or double (“) quotes called delimiters.

## ****Data Types in R Programming****

The fundamental/atomic data types in R programming are as follows:

* Numeric
* Integer
* Complex
* Character
* Logical

Elements of these R programming data types are often combined to form data structures.

### ****Numeric Data Type****

In R, if we assign any decimal value to a variable it becomes a variable of a numeric data type.  
For example, the statement below assigns a numeric data type to the variable “x”.

x = 45.6

And, the following statement is used to print the data type of the variable “x”:

class(x)

Output:- [1] "numeric"

### ****Integer Data Type****

To create an integer variable in R, we need to call the (as.Integer) function while assigning value to a variable.  
For example:-

e = as.integer(3)

class(e)

Output: [1] "integer"

Another way of creating an integer variable is by using the L keyword as follows:

x = 5L

class(x)

Output: [1] "integer"

### ****Complex Data Type****

The values containing the imaginary number ‘i’ (iota) are called complex values.  
The following code gives an error when run:

sqrt(−1)

Output:[1] NaN

Warning message:

In sqrt(−1) : NaNs produced

To overcome this error, we coerce the value (−1) into a complex value and denote it as ‘i’.

sqrt(as.complex(−1)

Output:[1] 0+1i

For example:

z = 2 + 3i

### ****Character Data Type****

This data type is used to represent strings.  
For example:

str1 = "Sam"

class(str1)

Output: [1] "character"

We can also use the **as.character()**function to convert objects into character values.  
For example:

x = as.character(55.7)

print(x)

Output:[1] "55.7"

class(x)

Output:[1] "character"

### ****Logical Data Type****

A logical data type stores either of the two values: TRUE -/FALSE. A logical value is often generated when two values are compared.  
For example:-

x = 3

y = 5

a = x > y

a

Output:

FALSE

Three standard logical operations,i.e., AND(&), OR(|), and NOT(!) yield a variable of the logical data type.  
For example:-

x= TRUE; y = FALSE

x & y

Output:

[1] FALSE

x | y

Output:

[1] TRUE

!x

Output:

[1] FALSE

**What are Vectors?**

Vectors are the basic R data objects and there are 6 types of the atomic vectors. They can be

* Integer,
* Logical,
* Double,
* Complex,
* Character and
* Raw

**Creation of Vector**

There are two types of vector creation:

* Single Element Vector
* Multiple Elements Vector

### ****Single Element Vector****

Whenever 1 word is written in R, it becomes a vector of length 1 and fits in one of the above vector types.

#Atomic vector of integer type

print(52L)

#Logical type

print(TRUE)

**Output:**

sol <- nchar("Counting number of

[1] 52

[1] TRUE

## ****Multiple Elements Vector****

### ****Using Colon operator with numeric data****

This operator helps in a constant change over the numeric data with limits.

**Example:**

#Creating sequence

a<- 4:10

b<-2.2:4.2

print(a)

print(b)

**Output:**

[1] 4 5 6 7 8 9 10

[1] 2.2 3.2 4.2

### ****Using sequence(Seq.) operator****

#Creating vector by incrementing by 0.2

print(seq(2, 3, by = 0.2))

**Output:**

[1]  2.0 2.2 2.4 2.6 2.8 3.0

### ****Accessing Vector Elements****

Indexing helps access the elements of a vector. The[ ] brackets are used for indexing.

Indexing starts with number 1 position. A negative value in the index rejects that element from output.  0 and 1 or TRUE and FALSE can be used for indexing.

#accessing vector elements

x<- c("letter one", "letter two", "letter three", "four", "five", "six")

b<- x[c(1,3,6)]

print(b)

#Usage of  logical Index

d<- x[c(FALSE, FALSE,TRUE,TRUE,FALSE)]

print(d)

#Using negative indexing

e<- x[c(-1,-2,-3,-4)]

print(e)

**Output:**

[1] "letter one" "letter three" "six"

[1] "letter three" "four"

[1] "five" "six"

## ****Vector Manipulation****

### ****Vector Arithmetic****

Two vectors having the same length can do arithmetic operations such as addition, subtraction, multiplication, and division to get vector output.

### ****Vector Element Recycling****

When applying arithmetic operations to two vectors of unequal length, the elements of the shorter vector are recycled to complete the operations.

**Example:**

a <- c(2,4,6,8)

b <- c(3,8)

#b becomes c(3,8,3,8)

add.op <- a+b

print(add.op)

**Output:**

[1]  5  12  9  16

### ****Vector Element Sorting****

Sorting of elements in a vector takes place in ascending or descending order. It can be either numbers or characters.

**Example:**

a <- c(2, 5, -6, 0)

#sorting elements of vector

sort.sol <- sort(a)

print(sort.sol)

#sorting character vectors in decreasing order

b <- c("Blue", "Red", "Green")

revsort.sol <- sort(b, decreasing = TRUE)

print(revsort.sol)

**Output:**

[1] -6 0 2 5

[1] "Blue" "Green" "Red"

## ****What are Lists?****

Lists are the R objects with numbers, strings, vectors and another list or matrix inside it.

## ****Creating  a List****

Example to create a list containing numbers, strings, vectors, and logical values.

#creating a list

list\_info <- list("Blue", "Yellow", c(12, 13, 14), TRUE, 13.12, 103.4)

print(list\_info)

**Output:**

[[1]]

[1] "Blue"

[[2]]

[1] "Yellow"

[[3]]

[1] 12 13 14

[[4]]

[1] TRUE

[[5]]

[1] 13.12

[[6]]

[1] 103.4

## ****Naming List Elements****

Names can be given to list elements and can be accessed using the corresponding names.

**Example:**

#Creating  a list which contains a matrix and a vector

list\_name <- list(matrix(c(1,2,3,4,5,6), nrow = 2), c("mon","tue","wed"))

#Naming elements in the list

names(list\_name) <- c("Matrix", "half\_week")

#displaying list

print(list\_name)

**Output:**

$Matrix

     [,1] [,2] [,3]

[1,]    1    3   5

[2,]    2    4   6

$ half\_week

[1] "mon" "tue" "wed"

## ****Accessing List Elements****

Index of the element of the list can be given access to Elements of the list.

**Syntax:**

list\_name <- list(.,..,.)

names(list\_name) <- c(.,.,.)

print(list\_name[1])

## ****Manipulating List Elements****

Addition, subtraction or deleting and updating the list elements can be done;

**Few examples are:**

#Creating a list which contains a vector, a matrix and a list

list\_name <- list(c("Mon", Tue", "Wed"), matrix(c(2,1,1,1,5,6), nrow =2), list("milk", 1.2)

#Naming elements in the list

names(list\_name) <- c("half week", "Matrix", "A simple list")

#Creating an element at the end of list

list\_name [4] <- "An Element"

print(list\_name[4])

#Withdrawing the last element

list\_name[4] <-NULL

#Output last element

print(list\_name[4])

**Output:**

[[1]]

[1] " An Element

$

NULL

## ****Merging Lists****

Merging can be done by placing all lists into one list() function.

**Example:**

#Creating lists

lista <- list(2,4,6)

listb <- list("Jan", "Feb", "Mar")

#Merging lists

merge.list <- c(lista. listb)

#output merged list

print(merge.list)

**Output:**

[[1]]

[1] 2

[[2]]

[1] 4

[[3]]

[1] 6

[[4]]

[1] "Jan"

[[5]]

[1] "Feb"

[[6]]

[1] "Mar"

## ****Converting List to Vector****

Using unlist() function we can convert a list to a vector so that all the elements of the vector can be used for further manipulation such as applying arithmetic operations.

Example:

#Creating lists

lista <- list(1:3)

listb <- list(4:6)

#Converting lists to vector

cva <- unlist(lista)

cvb <- unlist(listb)

print(cva)

print(cvb)

**Output:**

[1] 1 2 3

[1] 4 5 6

**R Arithmetic Operators**

These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

|  |  |
| --- | --- |
| Arithmetic Operators in R | |
| Operator | Description |
| + | Addition |
| – | Subtraction |
| \* | Multiplication |
| / | Division |
| ^ | Exponent |
| %% | Modulus (Remainder from division) |
| %/% | Integer Division |

An example run

> x <- 5

> y <- 16

> x+y

[1] 21

> x-y

[1] -11

> x\*y

[1] 80

> y/x

[1] 3.2

> y%/%x

[1] 3

> y%%x

[1] 1

> y^x

[1] 1048576

**R Logical Operators**

Logical operators are used to carry out Boolean operations like AND, OR etc.

|  |  |
| --- | --- |
| Logical Operators in R | |
| Operator | Description |
| ! | Logical NOT |
| & | Element-wise logical AND |
| && | Logical AND |
| | | Element-wise logical OR |
| || | Logical OR |

Operators & and | perform element-wise operation producing result having length of the longer operand.

But && and || examines only the first element of the operands resulting into a single length logical vector.

Zero is considered FALSE and non-zero numbers are taken as TRUE. An example run.

> x <- c(TRUE,FALSE,0,6)

> y <- c(FALSE,TRUE,FALSE,TRUE)

> !x

[1] FALSE TRUE TRUE FALSE

> x&y

[1] FALSE FALSE FALSE TRUE

> x&&y

[1] FALSE

> x|y

[1] TRUE TRUE FALSE TRUE

> x||y

[1] TRUE

## Matrix Operations in R

#### The Matrix

**# the matrix function**

**# R wants the data to be entered by columns starting with column one**

**# 1st arg: c(2,3,-2,1,2,2) the values of the elements filling the columns**

**# 2nd arg: 3 the number of rows**

**# 3rd arg: 2 the number of columns**

**> A <- matrix(c(2,3,-2,1,2,2),3,2)**

**> A**

[,1] [,2]

[1,] 2 1

[2,] 3 2

[3,] -2 2

#### Is Something a Matrix

**> is.matrix(A)**

[1] TRUE

**> is.vector(A)**

[1] FALSE

#### Multiplication by a Scalar

**> c <- 3**

**> c\*A**

[,1] [,2]

[1,] 6 3

[2,] 9 6

[3,] -6 6

#### Matrix Addition & Subtraction

**> B <- matrix(c(1,4,-2,1,2,1),3,2)**

**> B**

[,1] [,2]

[1,] 1 1

[2,] 4 2

[3,] -2 1

**> C <- A + B**

**> C**

[,1] [,2]

[1,] 3 2

[2,] 7 4

[3,] -4 3

**> D <- A - B**

**> D**

[,1] [,2]

[1,] 1 0

[2,] -1 0

[3,] 0 1

#### Matrix Multiplication

**> D <- matrix(c(2,-2,1,2,3,1),2,3)**

**> D**

[,1] [,2] [,3]

[1,] 2 1 3

[2,] -2 2 1

**> C <- D %\*% A**

**> C**

[,1] [,2]

[1,] 1 10

[2,] 0 4

**> C <- A %\*% D**

**> C**

[,1] [,2] [,3]

[1,] 2 4 7

[2,] 2 7 11

[3,] -8 2 -4

**> D <- matrix(c(2,1,3),1,3)**

**> D**

[,1] [,2] [,3]

[1,] 2 1 3

**> C <- D %\*% A**

**> C**

[,1] [,2]

[1,] 1 10

**> C <- A %\*% D**

Error in A %\*% D : non-conformable arguments

#### Transpose of a Matrix

**> AT <- t(A)**

**> AT**

[,1] [,2] [,3]

[1,] 2 3 -2

[2,] 1 2 2

**> ATT <- t(AT)**

**>ATT**

[,1] [,2]

[1,] 2 1

[2,] 3 2

[3,] -2 2

# R - Data Frames

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

* The column names should be non-empty.
* The row names should be unique.
* The data stored in a data frame can be of numeric, factor or character type.
* Each column should contain same number of data items.
* Create Data Frame
* [Live Demo](http://tpcg.io/k9JHFY)
* # Create the data frame.
* emp.data <- data.frame(
* emp\_id = c (1:5),
* emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),
* salary = c(623.3,515.2,611.0,729.0,843.25),
* start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
* "2015-03-27")),
* stringsAsFactors = FALSE
* )
* # Print the data frame.
* print(emp.data)
* When we execute the above code, it produces the following result −
* emp\_id emp\_name salary start\_date
* 1 1 Rick 623.30 2012-01-01
* 2 2 Dan 515.20 2013-09-23
* 3 3 Michelle 611.00 2014-11-15
* 4 4 Ryan 729.00 2014-05-11
* 5 5 Gary 843.25 2015-03-27

## Get the Structure of the Data Frame

The structure of the data frame can be seen by using **str()** function.

# Get the structure of the data frame.

str(emp.data)

When we execute the above code, it produces the following result −

'data.frame': 5 obs. of 4 variables:

$ emp\_id : int 1 2 3 4 5

$ emp\_name : chr "Rick" "Dan" "Michelle" "Ryan" ...

$ salary : num 623 515 611 729 843

$ start\_date: Date, format: "2012-01-01" "2013-09-23" "2014-11-15" "2014-05-11"

## Summary of Data in Data Frame

The statistical summary and nature of the data can be obtained by applying **summary()** function.

# Print the summary.

print(summary(emp.data))

When we execute the above code, it produces the following result −

emp\_id emp\_name salary start\_date

Min. :1 Length:5 Min. :515.2 Min. :2012-01-01

1st Qu.:2 Class :character 1st Qu.:611.0 1st Qu.:2013-09-23

Median :3 Mode :character Median :623.3 Median :2014-05-11

Mean :3 Mean :664.4 Mean :2014-01-14

3rd Qu.:4 3rd Qu.:729.0 3rd Qu.:2014-11-15

Max. :5 Max. :843.2 Max. :2015-03-27

## Extract Data from Data Frame

Extract specific column from a data frame using column name.

# Extract Specific columns.

result <- data.frame(emp.data$emp\_name,emp.data$salary)

print(result)

When we execute the above code, it produces the following result −

emp.data.emp\_name emp.data.salary

1 Rick 623.30

2 Dan 515.20

3 Michelle 611.00

4 Ryan 729.00

5 Gary 843.25

## Expand Data Frame

A data frame can be expanded by adding columns and rows.

### Add Column

Just add the column vector using a new column name.

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

stringsAsFactors = FALSE

)

# Add the "dept" coulmn.

emp.data$dept <- c("IT","Operations","IT","HR","Finance")

v <- emp.data

print(v)

When we execute the above code, it produces the following result −

emp\_id emp\_name salary start\_date dept

1 1 Rick 623.30 2012-01-01 IT

2 2 Dan 515.20 2013-09-23 Operations

3 3 Michelle 611.00 2014-11-15 IT

4 4 Ryan 729.00 2014-05-11 HR

5 5 Gary 843.25 2015-03-27 Finance

### Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the **rbind()** function.

In the example below we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

# Create the first data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

dept = c("IT","Operations","IT","HR","Finance"),

stringsAsFactors = FALSE

)

# Create the second data frame

emp.newdata <- data.frame(

emp\_id = c (6:8),

emp\_name = c("Rasmi","Pranab","Tusar"),

salary = c(578.0,722.5,632.8),

start\_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),

dept = c("IT","Operations","Fianance"),

stringsAsFactors = FALSE

)

# Bind the two data frames.

emp.finaldata <- rbind(emp.data,emp.newdata)

print(emp.finaldata)

When we execute the above code, it produces the following result −

emp\_id emp\_name salary start\_date dept

1 1 Rick 623.30 2012-01-01 IT

2 2 Dan 515.20 2013-09-23 Operations

3 3 Michelle 611.00 2014-11-15 IT

4 4 Ryan 729.00 2014-05-11 HR

5 5 Gary 843.25 2015-03-27 Finance

6 6 Rasmi 578.00 2013-05-21 IT

7 7 Pranab 722.50 2013-07-30 Operations

8 8 Tusar 632.80 2014-06-17 Fianance

# R - Functions

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows −

function\_name <- function(arg\_1, arg\_2, ...) {

Function body

}

## Function Components

The different parts of a function are −

* **Function Name** − This is the actual name of the function. It is stored in R environment as an object with this name.
* **Arguments** − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
* **Function Body** − The function body contains a collection of statements that defines what the function does.
* **Return Value** − The return value of a function is the last expression in the function body to be evaluated.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

## Built-in Function

Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc. They are directly called by user written programs. You can refer [most widely used R functions.](https://cran.r-project.org/doc/contrib/Short-refcard.pdf)

[Live Demo](http://tpcg.io/pcfp4i)

# Create a sequence of numbers from 32 to 44.

print(seq(32,44))

# Find mean of numbers from 25 to 82.

print(mean(25:82))

# Find sum of numbers frm 41 to 68.

print(sum(41:68))

When we execute the above code, it produces the following result −

[1] 32 33 34 35 36 37 38 39 40 41 42 43 44

[1] 53.5

[1] 1526

## User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

# Create a function to print squares of numbers in sequence.

new.function <- function(a) {

for(i in 1:a) {

b <- i^2

print(b)

}

}

## Calling a Function

[Live Demo](http://tpcg.io/WxqnYt)

# Create a function to print squares of numbers in sequence.

new.function <- function(a) {

for(i in 1:a) {

b <- i^2

print(b)

}

}

# Call the function new.function supplying 6 as an argument.

new.function(6)

When we execute the above code, it produces the following result −

[1] 1

[1] 4

[1] 9

[1] 16

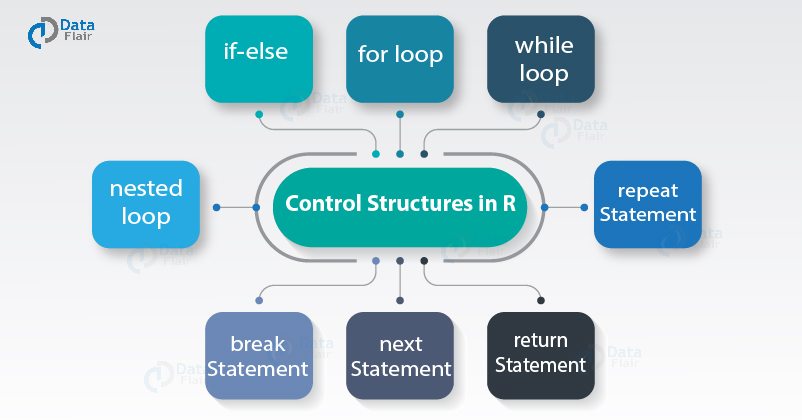
[1] 25

[1] 36

# Control Structures in R

In order to control the execution of the expressions flow in R, we make use of the control structures. These control structures are also called as loops in R. There are eight types of control structures in R:

* if
* if-else
* for
* nested loops
* while
* repeat and break
* next
* return



### 1. if Condition in R

This task is carried out only if this condition is returned as TRUE. R makes it even easier: You can drop the word then and specify your choice in an if statement.

**Syntax:**

1. **if** (test\_expression) {
2. statement
3. }

### 2. if-else Condition in R

An if…else statement contains the same elements as an if statement (see the preceding section), with some extra elements:

* The keyword else, placed after the first code block.
* The second block of code, contained within braces, that has to be carried out, only if the result of the condition in the if() statement is FALSE.

**Syntax:**

1. **if** (test\_expression) {
2. statement
3. } else {
4. statement

### 3. for Loop in R

A loop is a sequence of instructions that is repeated until a certain condition is reached. for, while and repeat, with the additional clauses break and next are used to construct loops.

**Example:**

These control structures in [R](https://en.wikipedia.org/wiki/R_(programming_language)), made of the rectangular box ‘init’ and the diamond. It is executed a known number of times. for is a block that is contained within curly braces.

1. values <- **c**(1,2,3,4,5)
2. **for**(id in 1:5){
3. **print**(values[id])
4. }

### 4. Nested Loop in R

It is similar to the standard for **loop**, which makes it easy to convert **for** **loop** to a **foreach loop**. Unlike many parallel programming packages for R, foreach doesn’t require the body of for loop to be turned into a function. We can call this a nesting operator because it is used to create nested foreach loops.

**Example:**

1. mat <- **matrix**(1:10, 2)
2. **for** (id1 in **seq**(**nrow**(mat))) {
3. **for** (id2 in **seq**(**ncol**(mat))) {
4. **print**(mat[id1, id2])
5. }
6. }

### 5. while Loop in R

The format is**while(cond) expr**, where **cond** is the condition to test and **expr** is an expression.

R would complain about the missing expression that was supposed to provide the required True or False and in fact, it does not know ‘response’ before using it in the loop. We can also do this because, if we answer right at first attempt, the loop will not be executed at all.

**Example:**

1. val = 2.987
2. **while**(val <= 4.987) {
3. val = val + 0.987
4. **print**(**c**(val,val-2,val-1))
5. }

### 6. repeat and break Statement in R

We use **break** statement inside a loop (repeat, for, while) to stop the iterations and flow the control outside of the loop. While in a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

A **repeat** loop is used to iterate over a block of code, multiple numbers of times. There is no condition check in a repeat loop to exit the loop. We ourselves put a condition explicitly inside the body of the loop and use the break statement to exit the loop. Failing to do so will result in an infinite loop.

**Syntax:**

1. repeat {
2. # simulations; generate some value have an expectation if within some range,
3. # then exit the loop
4. **if** ((value - expectation) <= threshold) {
5. break
6. }
7. }

### 7. next Statement in R

**next** jumps to the next cycle without completing a particular iteration. In fact, it jumps to the evaluation of the condition holding the current loop. Next statement enables to skip the current iteration of a loop without terminating it.

**Example:**

1. x = 1: 4
2. **for** (i in x) {
3. **if** (i == 2) {
4. next
5. }
6. **print**(i)
7. }

### 8. return Statement in R

Many times, we will require some functions to do processing and return back the result. This is accomplished with the **return()** statement in R.

**Syntax:**

1. **return**(expression)

R Debug Functions

### 1. traceback()

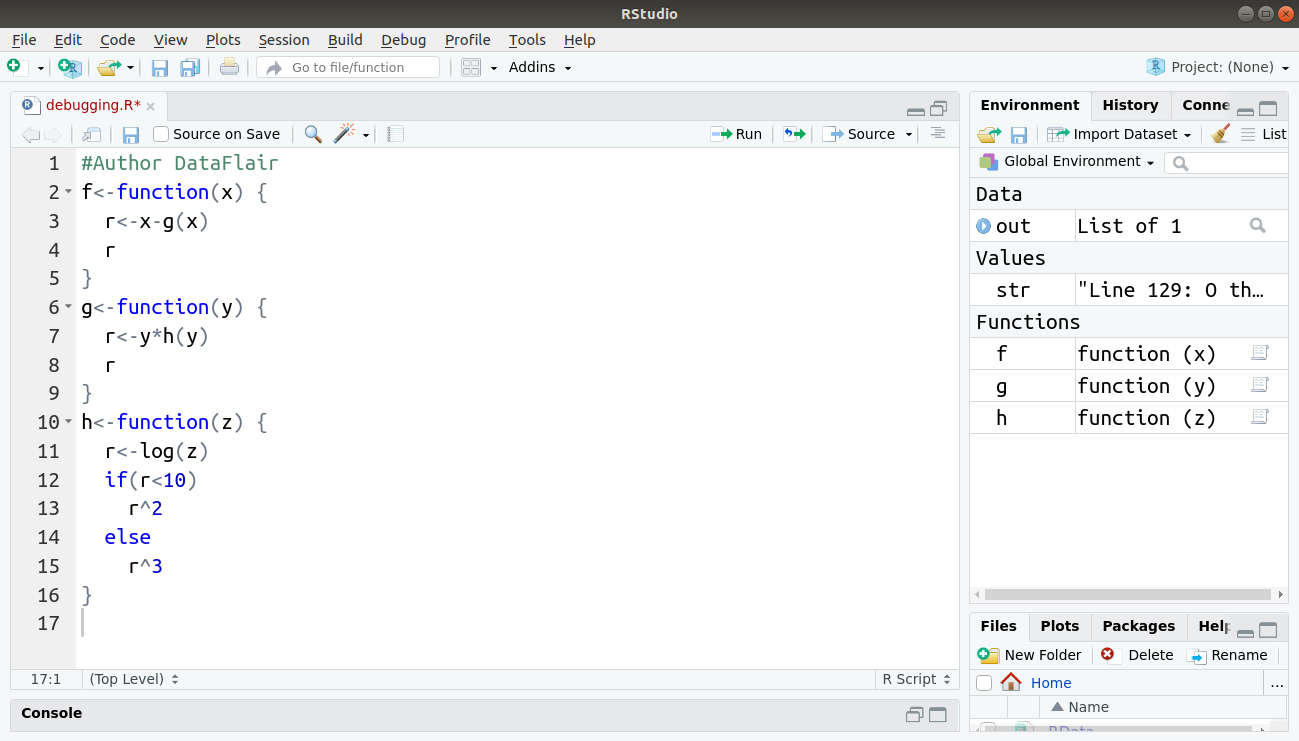
If our code has already crashed and we want to know where the offending line is, then try traceback(). This will (sometimes) show whereabouts in the code of the problem occurred.

When an R function fails, an error is printed to the screen. Immediately after the error, you can call traceback() to see in which function the error occurred. The traceback() function prints the list of functions that were called before the error occurred. The functions are printed in reverse order.

**For example:**

1. #Author DataFlair
2. f<-**function**(x) {
3. r<-x-**g**(x)
4. r
5. }
6. g<-**function**(y) {
7. r<-y\***h**(y)
8. r
9. }
10. h<-**function**(z) {
11. r<-**log**(z)
12. **if**(r<10)
13. r^2
14. else
15. r^3
16. }

**Code Display:**

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/09/traceback-input.jpg)

**Output:**

### [traceback output - R Debug](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/09/traceback-output.jpg)

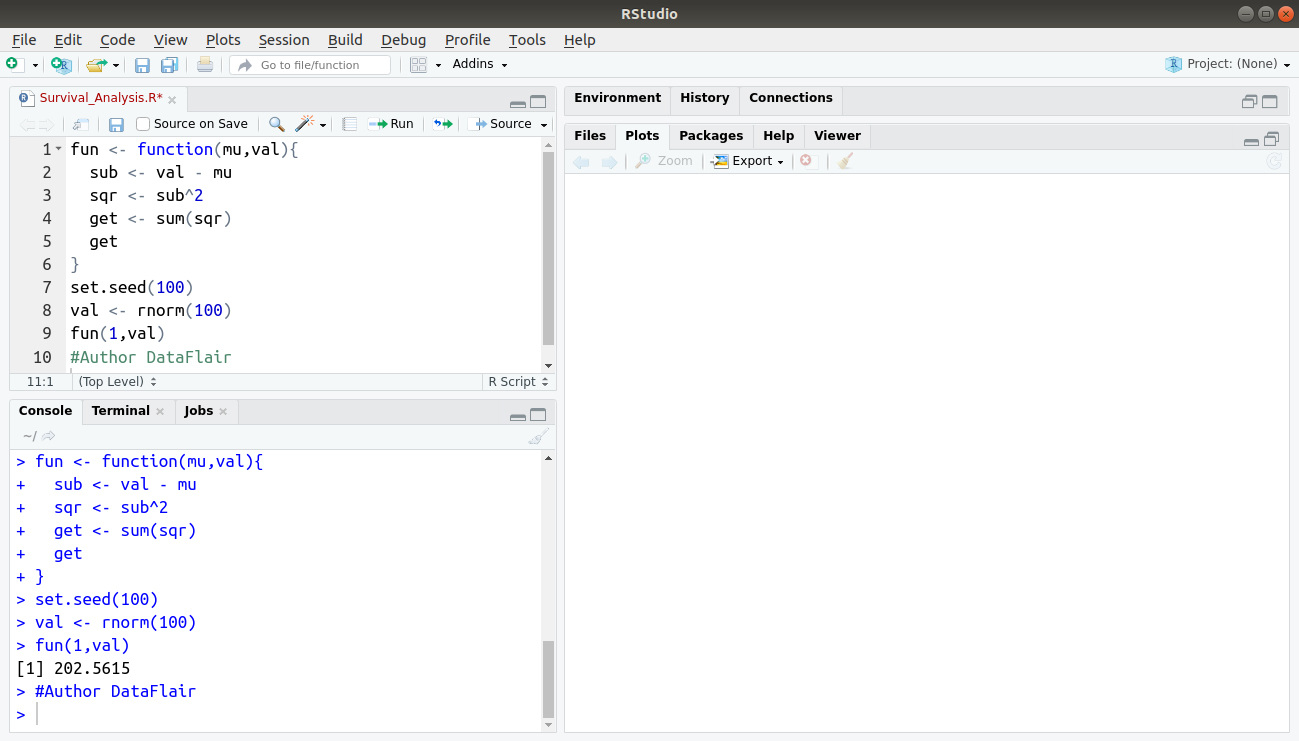
### 2. debug()

The function debug() in [R](https://www.r-project.org/news.html) allows the user to step through the execution of a function, line by line. At any point, we can print out values of variables or produce a graph of the results within the function. While debugging, we can simply type “c” to continue to the end of the current section of code. traceback() does not tell us where the error occurred in the function. In order to know which line causes the error, we will have to step through the function using debug().

**For example:**

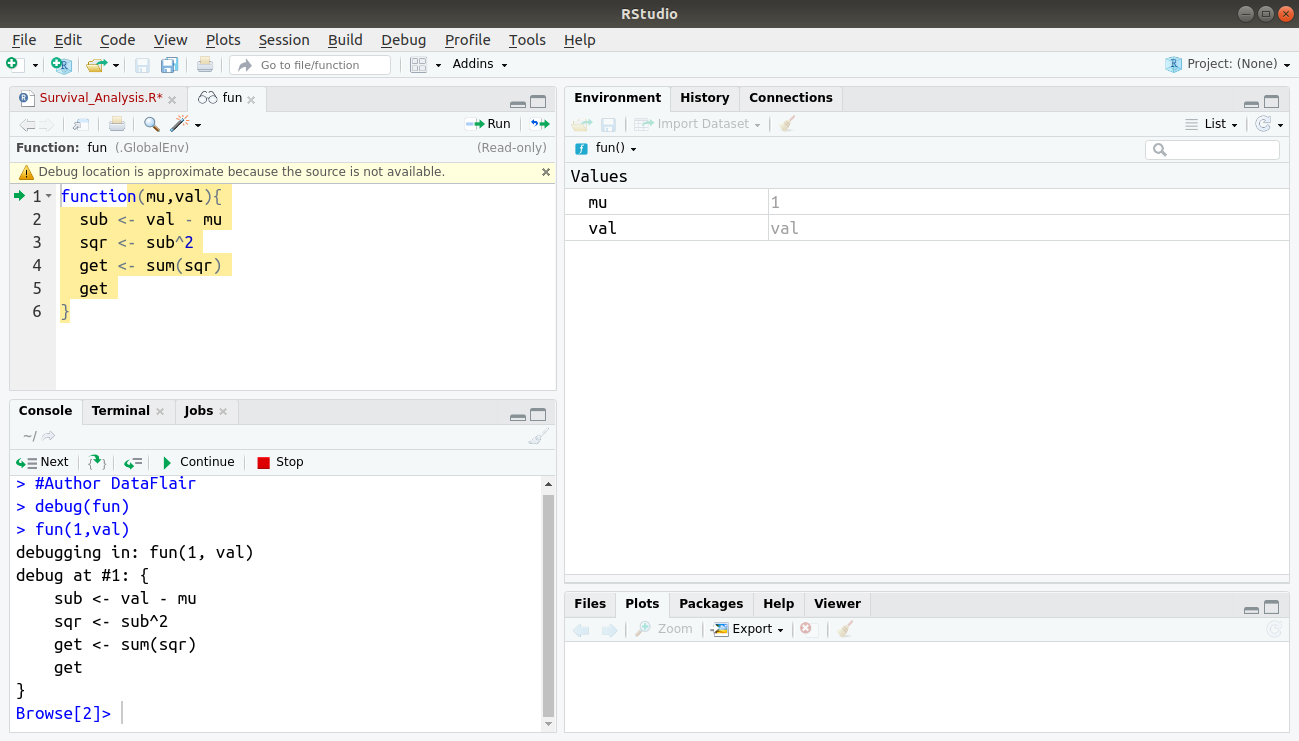
1. fun <- **function**(mu,val){
2. sub <- val - mu
3. sqr <- sub^2
4. get <- **sum**(sqr)
5. get
6. }
7. set.**seed**(100)
8. val <- **rnorm**(100)
9. **fun**(1,val)
10. #Author DataFlair

**Output:**

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/09/debug-input.jpg)

1. > #Author DataFlair
2. > **debug**(fun)
3. > **fun**(1,val)

**Output:**

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/09/debug-output.jpg)

After you see the “Browse[1]>” prompt, you can do different things:

* Typing n executes the current line and prints the next one;
* By typing Q, we can quit the debugging;
* Typing ‘where’ tells where you are in the function call stack;
* By typing ls(), we can list all objects in the local environment.

Typing an object name or print(<object name>) tells us current value of the object. If your object has name n, c or Q, we have to use print() to see their values.

***Do you know about***[***Object Oriented Programming in R***](https://data-flair.training/blogs/object-oriented-programming-in-r/)

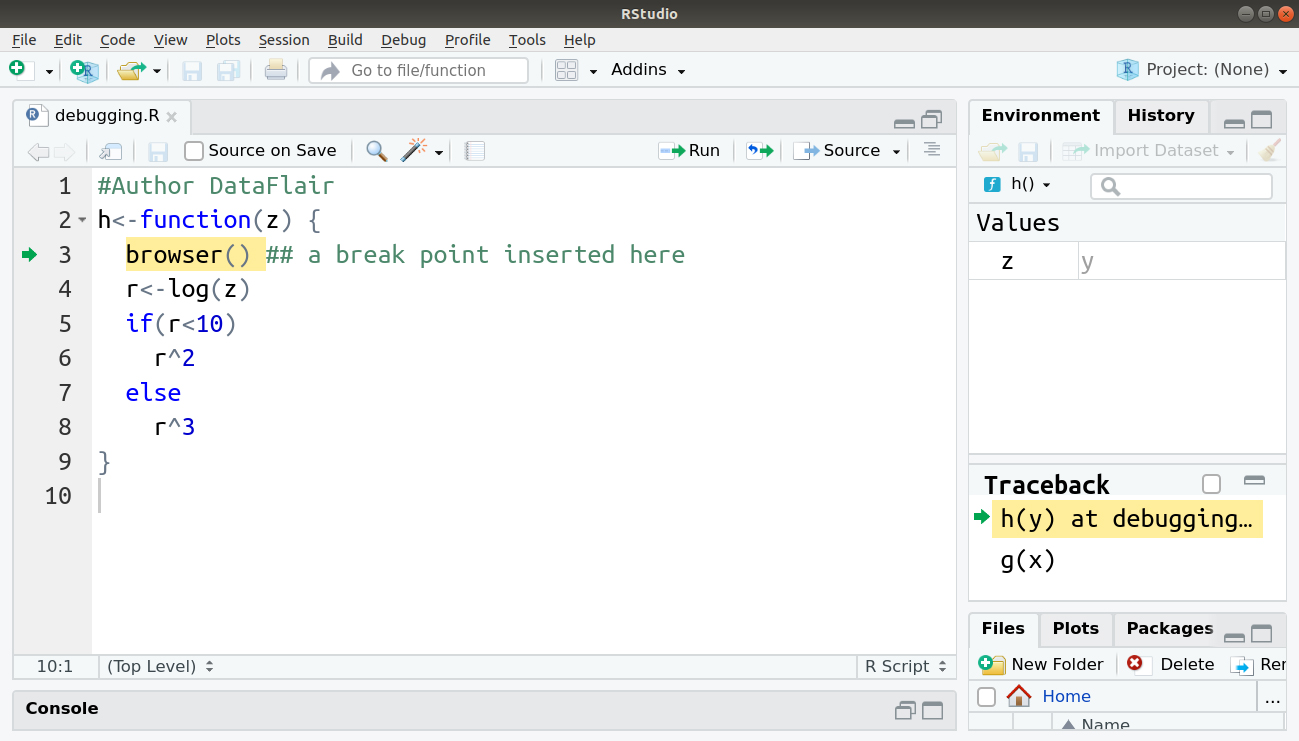
### 3. browser()

The R debug function browser() stops the execution of a function until the user allows it to continue. This is useful if we don’t want to step through the complete code, line-by-line, but we want it to stop at a certain point so we can check out what is going on. Inserting a call to the browser() in a function will pause the execution of a function at the point where the browser() is called. Similar to using debug() except we can control where execution gets paused.

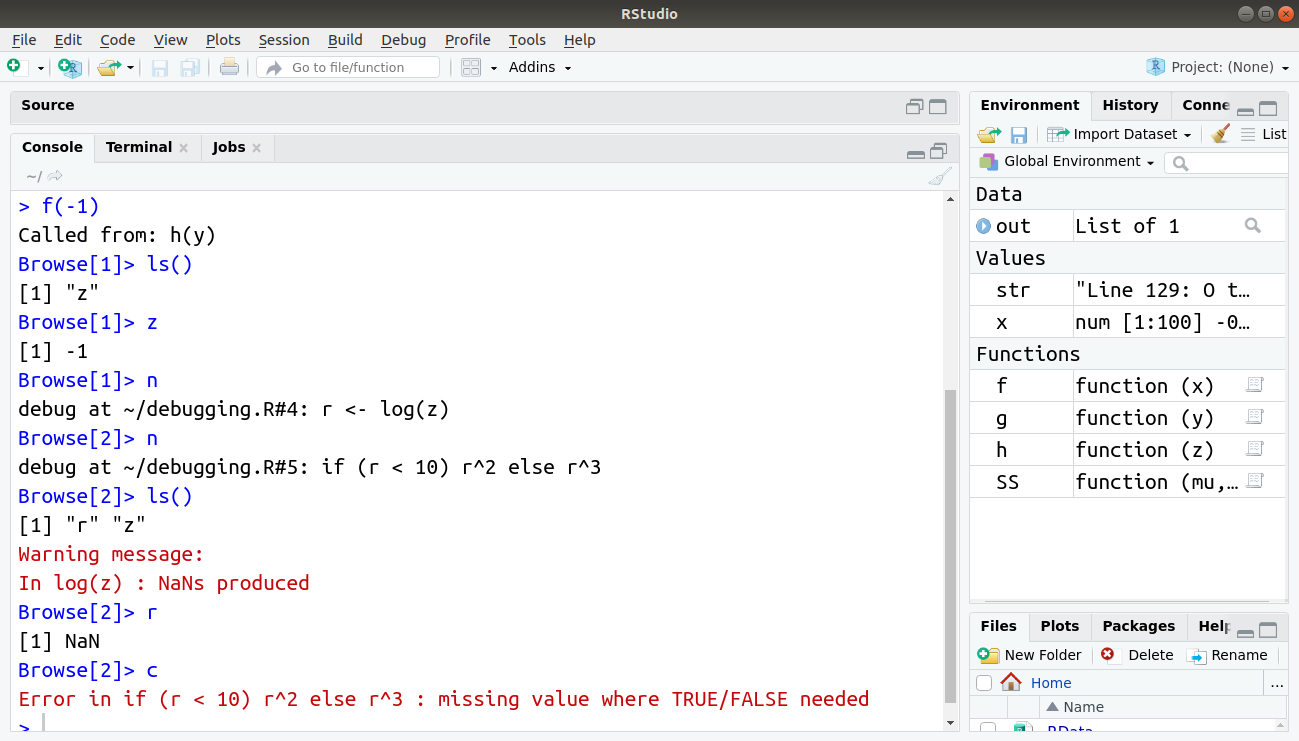
**For example:**

1. #Author DataFlair
2. h<-**function**(z) {
3. **browser**() ## a break point inserted here
4. r<-**log**(z)
5. **if**(r<10)
6. r^2
7. else
8. r^3
9. }

**Code Display:**

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/09/browser-input.jpg)

**Output:**

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/09/browser-output.jpg)

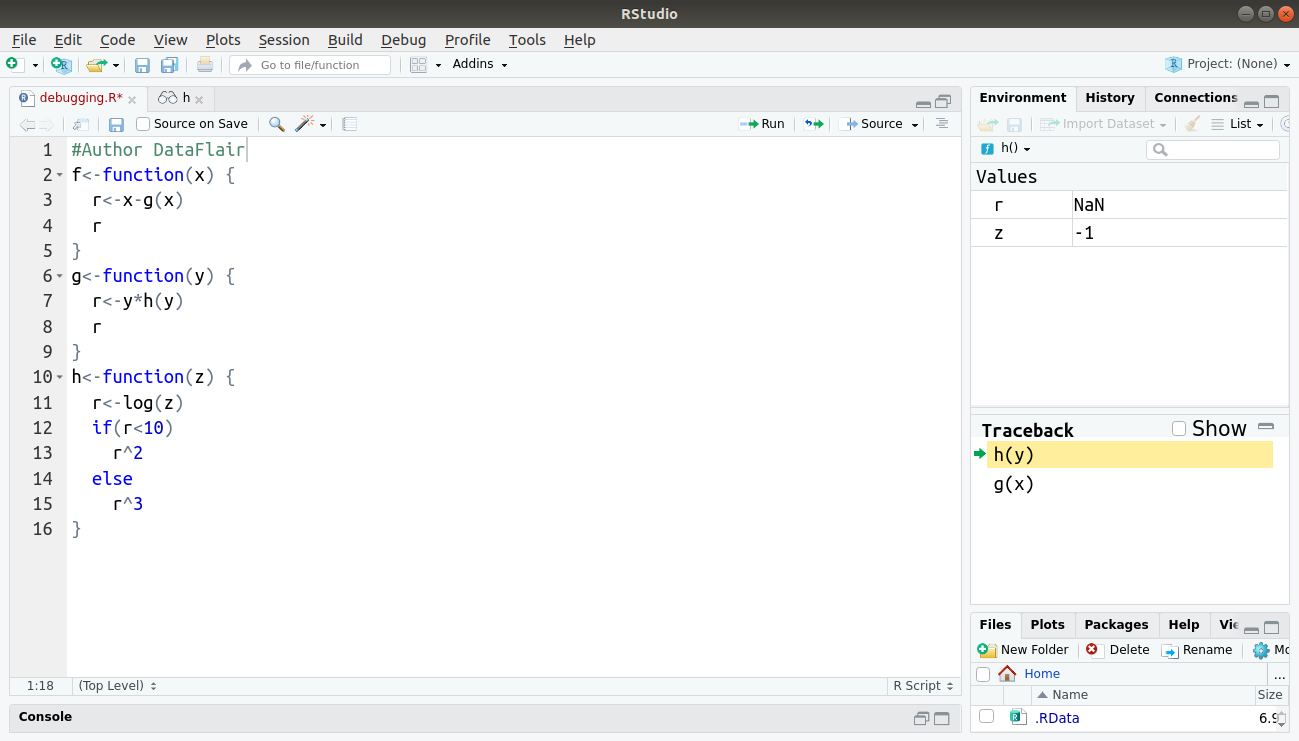
### 4. trace()

Calling trace() on a function allows the user to insert bits of code into a function. The syntax for R debug function trace() is a bit strange for first-time users. It might be better off using debug().

**For example:**

1. #Author DataFlair
2. f<-**function**(x) {
3. r<-x-**g**(x)
4. r
5. }
6. g<-**function**(y) {
7. r<-y\***h**(y)
8. r
9. }
10. h<-**function**(z) {
11. r<-**log**(z)
12. **if**(r<10)
13. r^2
14. else
15. r^3
16. }

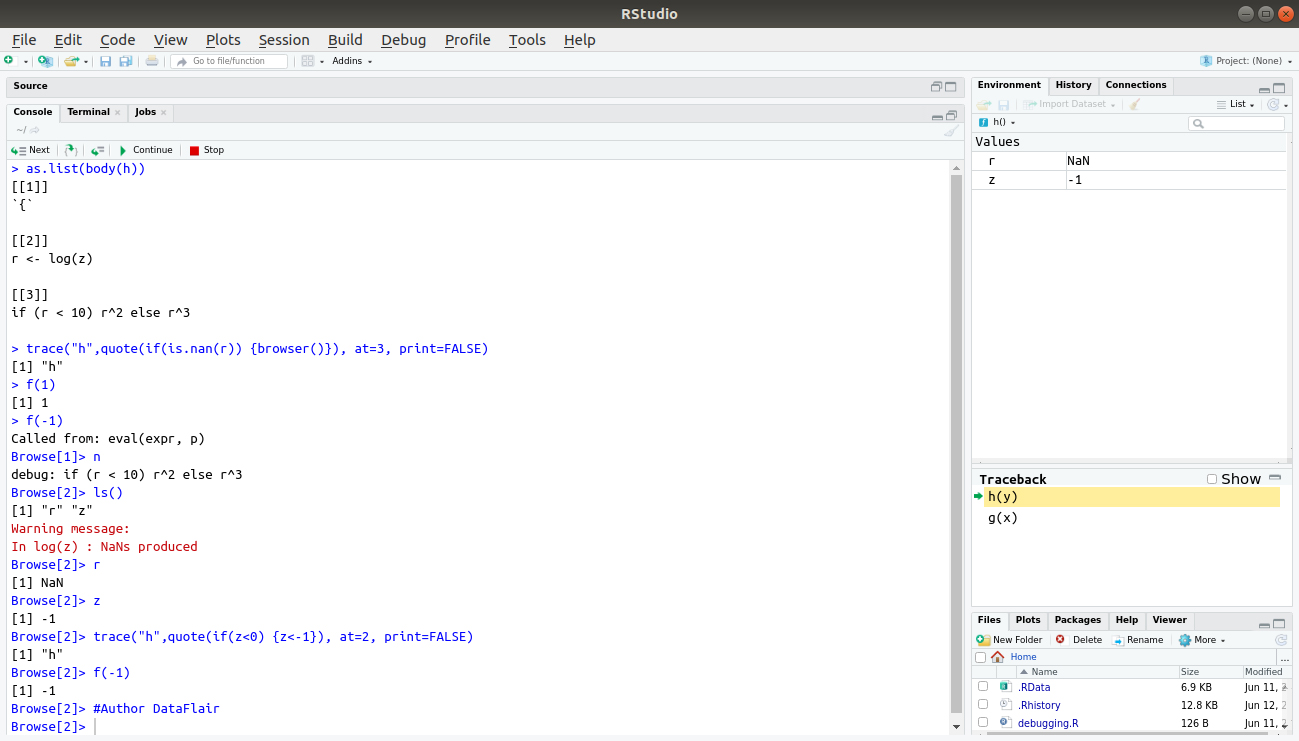
**Code Display:**

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/09/Trace-Input.jpg)

In the console, we type the following commands:

1. > as.**list**(**body**(h))
2. > **trace**("h",**quote**(**if**(is.**nan**(r)) {**browser**()}), at=3, print=FALSE)
3. > **f**(1)
4. > **f**(-1)
5. Browse[1]> n
6. Browse[2]> **ls**()
7. Warning message:
8. In **log**(z) : NaNs produced
9. Browse[2]> r
10. Browse[2]> z
11. Browse[2]> **trace**("h",**quote**(**if**(z<0) {z<-1}), at=2, print=FALSE)
12. Browse[2]> **f**(-1)

**Output:**

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/09/trace-output.jpg)

### 5. recover()

When we are debugging a function, recover() allows us to check variables in upper-level functions.

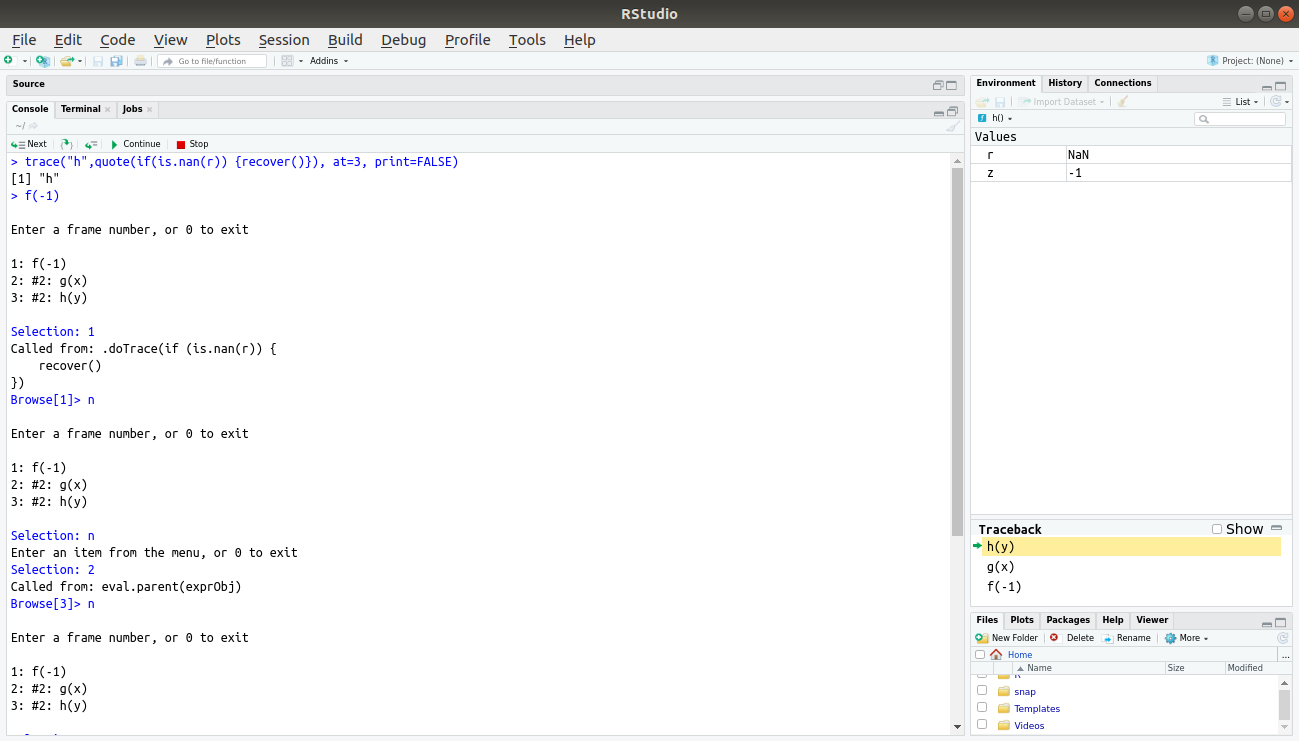
By typing a number in the selection, we are navigated to the function on the call stack and positioned in the browser environment.

* We can use recover() as an error handler, set using options() (e.g.options(error=recover)).
* When a function throws an error, execution is halted at the point of failure. We can browse the function calls and examine the environment to find the source of the problem.

In recover, we use the previous f(), g() and h() functions for debugging.

1. > **trace**("h",**quote**(**if**(is.**nan**(r)) {**recover**()}), at=3, print=FALSE)
2. > **f**(-1)
3. Selection: 1
4. Browse[1]> n
5. Selection: n
6. Selection: 2
7. Browse[3]> n
8. Selection: n
9. Selection: 3
10. Browse[5]> n
11. Selection: n
12. Selection: 0
13. Browse[4]>

**Output:**

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/09/Recover-Output.jpg)

By typing a number in the selection, we are navigated to the function on the call stack and we are positioned in the browser environment.