Data Analytics with R

BDS306C

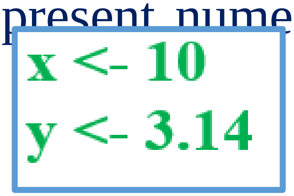
Module1 & 2 : Basics of R

Why R Programming Language?

- **Data Analysis and Visualization:** R is widely used for data analysis and visualization. Learning R will enable you to explore data, identify patterns, and create visualizations to communicate your findings effectively.
- **Statistical Analysis:** R is a powerful tool for statistical analysis. You can use it to run complex statistical models, perform hypothesis testing, and make inferences about data.
- **Machine Learning:** R has a wide range of packages that allow you to perform machine learning tasks, including classification, regression, clustering, and more. By learning R, you can build predictive models and analyze large datasets.
- **Reproducible Research:** R allows you to create reproducible research projects. By using RMarkdown, you can write code and text in a single document, making it easy to share your work with others and reproduce your results.
- **Programming Fundamentals:** Learning R will introduce you to programming fundamentals, such as data structures, control structures, functions, and object-oriented programming. These skills are transferable to other programming languages and will make you a more versatile programmer.

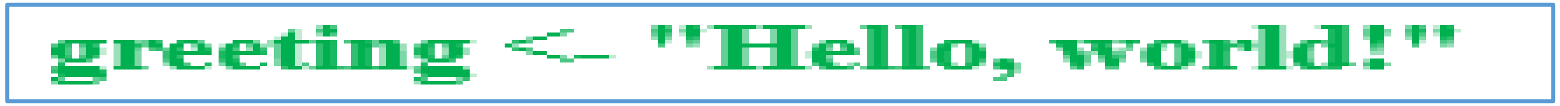
Data Types in R Language

- **Numeric:** Numeric data types represent numeric values. This data type can be used for both integer and floating-point values.



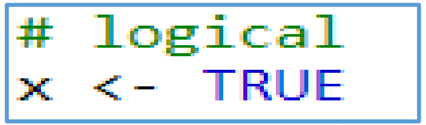
```
x <- 10
y <- 3.14
```

- **Character:** Character data types are used to represent text. In R, you can use single or double quotes to define character strings.



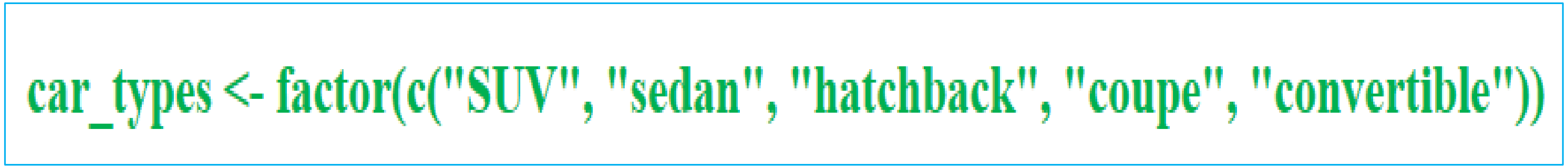
```
greeting <- "Hello, world!"
```

- **Logical:** Logical data types represent Boolean values, which can be either true or false.



```
# logical
x <- TRUE
```

- **Factor:** Factor data types are used to represent categorical data, such as the type of a car or the color of a dress. Factors are created using the factor() function.



```
car_types <- factor(c("SUV", "sedan", "hatchback", "coupe", "convertible"))
```

Data Types in R Language

- **Date:** Date data types are used to represent dates. In R, you can use the as.Date() function to convert a string to a date object.

```
today <- as.Date("2023-03-02")
```

- **Time:** Time data types are used to represent time. In R, you can use the as.POSIXct() function to convert a string to a time object.

```
now <- as.POSIXct("2023-03-02 15:30:00")
```

- **Complex:** Complex data types are used to represent complex numbers with real and imaginary parts.

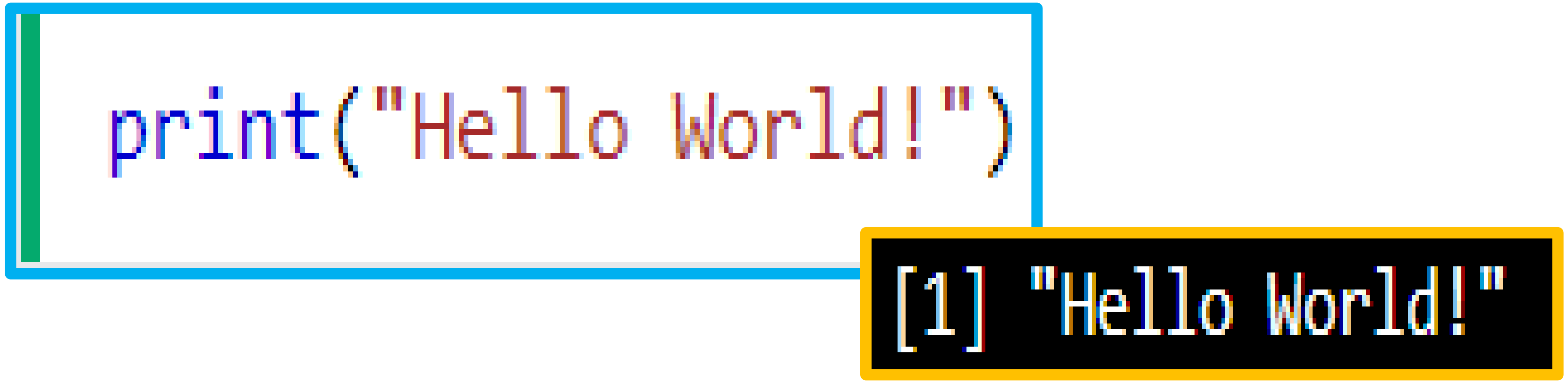
```
z <- 2+3i
```

- **Raw:** Raw data types are used to represent binary data, such as image or audio files

```
bin_data <- as.raw(c(0x48, 0x65, 0x6c, 0x6c, 0x6f))
```

Output Statement

- In R language, the print() function is used to display output to the console. When you run a command or expression in the R console or in an R script, the output is not automatically displayed. You need to use the print() function explicitly to display the output.



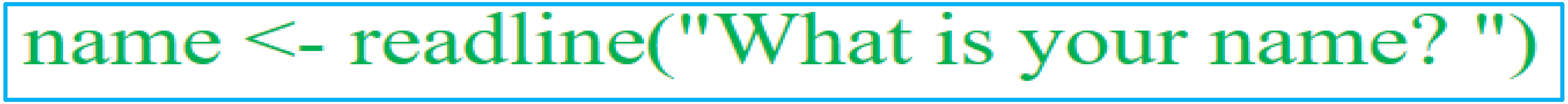
```
print("Hello World!")
```

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

Input Statement

- In R language, the readline() function is used to get user input from the console. This function prompts the user to enter a value and then waits for the user to type something and press enter.

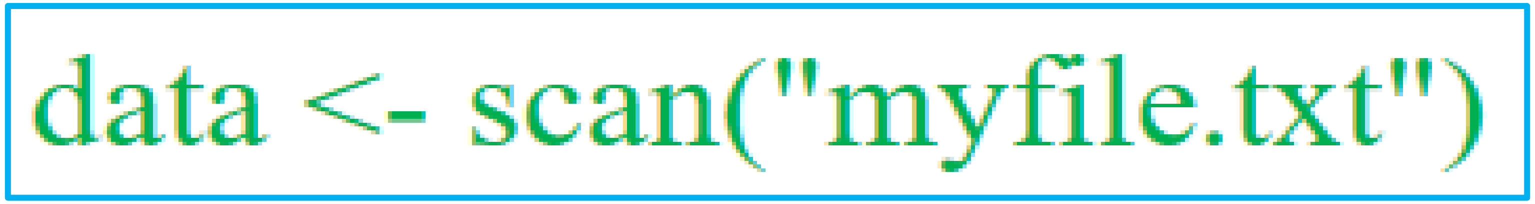
- In this example, the readline() function displays the prompt "What is your name?" in the console and waits for the user to enter a value. When the user types something and presses enter, the value is stored in the variable name.



```
name <- readline("What is your name? ")
```

scan() function

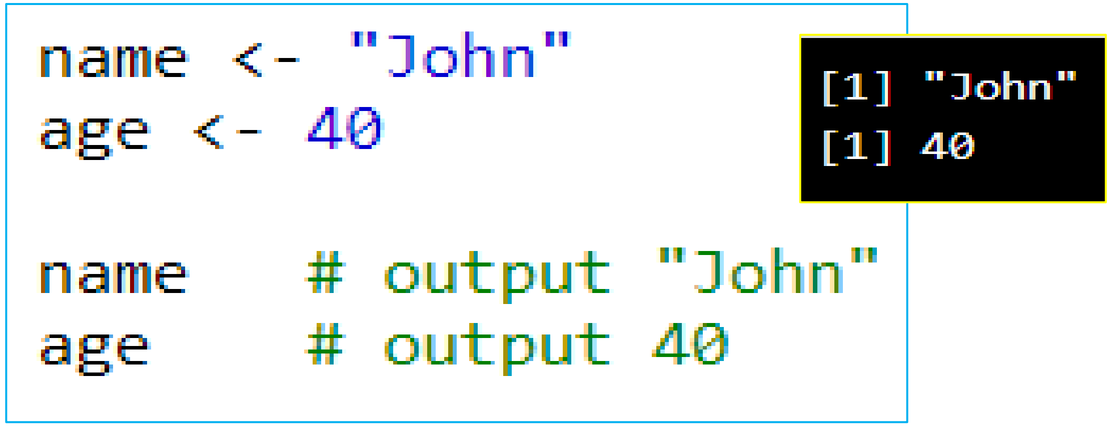
- You can also use the scan() function to read input from a file or from the clipboard. The scan() function reads input as a vector of values, with each value separated by whitespace or a specified delimiter.



```
data <- scan("myfile.txt")
```

Creating Variables in R

- Variables are containers for storing data values.
- R does not have a command for declaring a variable. A variable is created the moment you first assign a value to it. To assign a value to a variable, use the <- sign. To output (or print) the variable value, just type the variable name:



```
name <- "John"
age <- 40

name   # output "John"
age    # output 40
```

```
[1] "John"
[1] 40
```

Note : In other programming language, it is common to use = as an assignment operator. In R, we can use both = and <- as assignment operators.

Rules to Declare Variables

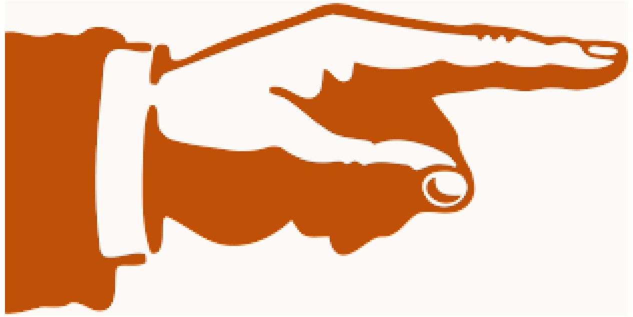
- Variable names should start with a letter and can contain letters, numbers, underscores, and periods.
- Variable names are case sensitive.
- Avoid using reserved keywords like if, else, for, while, etc., as variable names.
- Variable names should be meaningful and descriptive.
- Variables can be assigned values using the assignment operator <- or the equals sign =.
- You can declare multiple variables at once using the c() function.
- R language is dynamically typed, meaning you don't have to specify the data type of a variable when declaring it. The data type is automatically inferred based on the value assigned to it

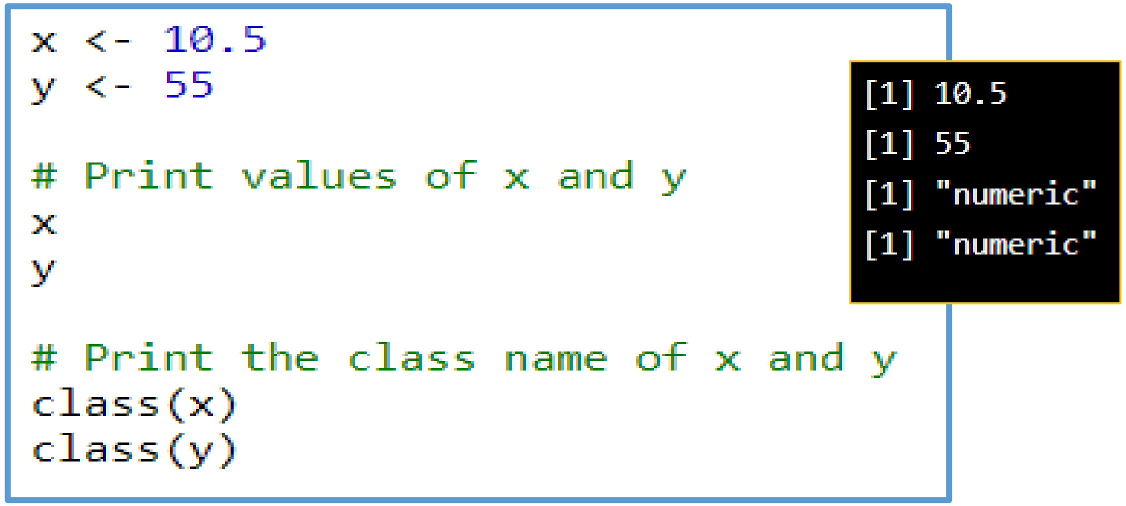
R Numbers

- There are three number types in R:
- numeric
- integer
- complex

numeric

- A numeric data type is the most common type in R, and contains any number with or without a decimal, like: 10.5, 55, 787:



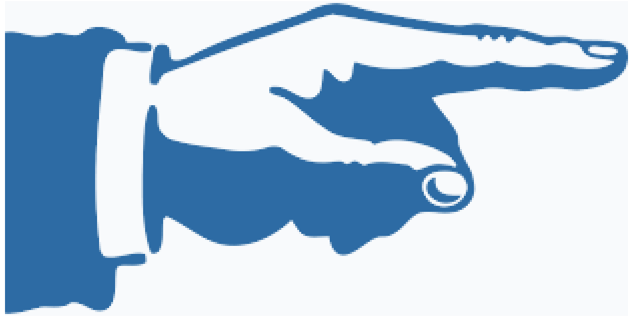


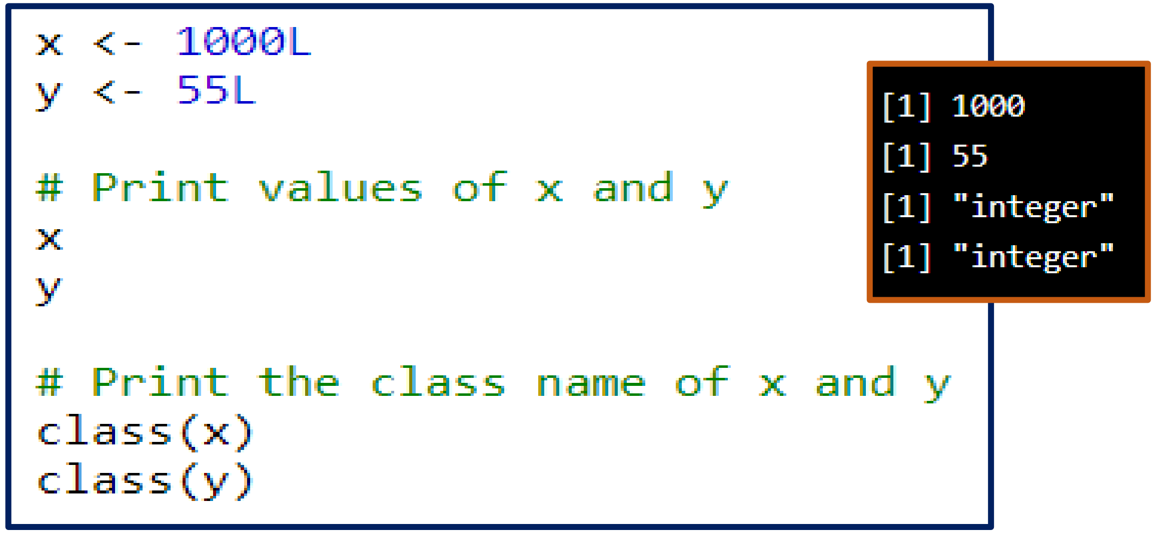
```
x <- 10.5
y <- 55

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

```
[1] 10.5
[1] 55
[1] "numeric"
[1] "numeric"
```

integer

- Integers are numeric data without decimals. This is used when you are certain that you will never create a variable that should contain decimals. To create an integer variable, you must use the letter L after the integer value:



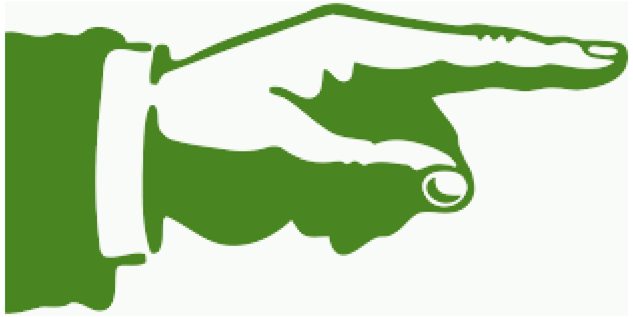


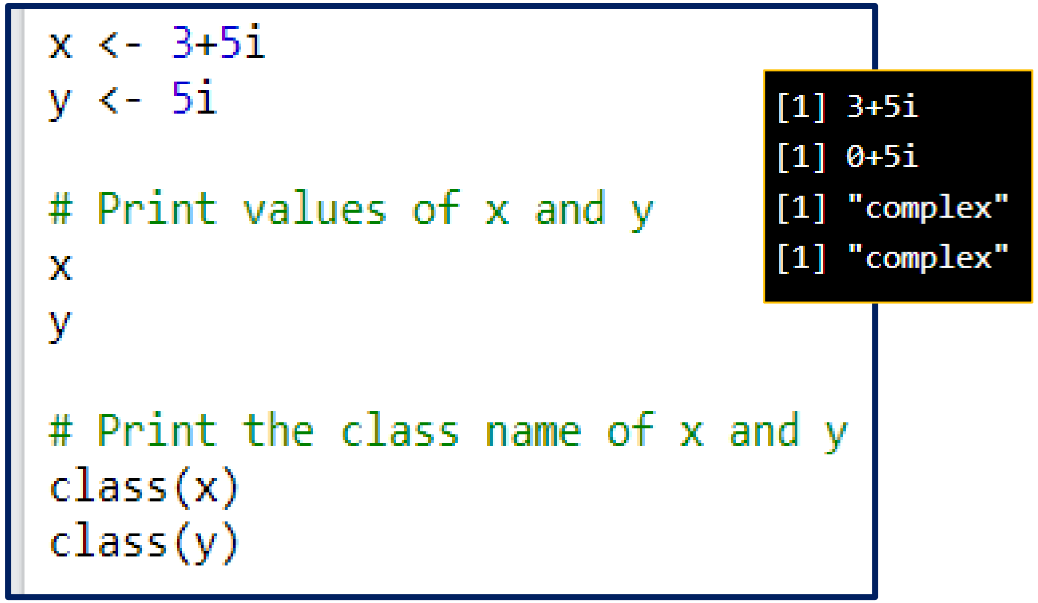
```
x <- 1000L
y <- 55L

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

```
[1] 1000
[1] 55
[1] "integer"
[1] "integer"
```

complex

- A complex number is written with an "i" as the imaginary part:



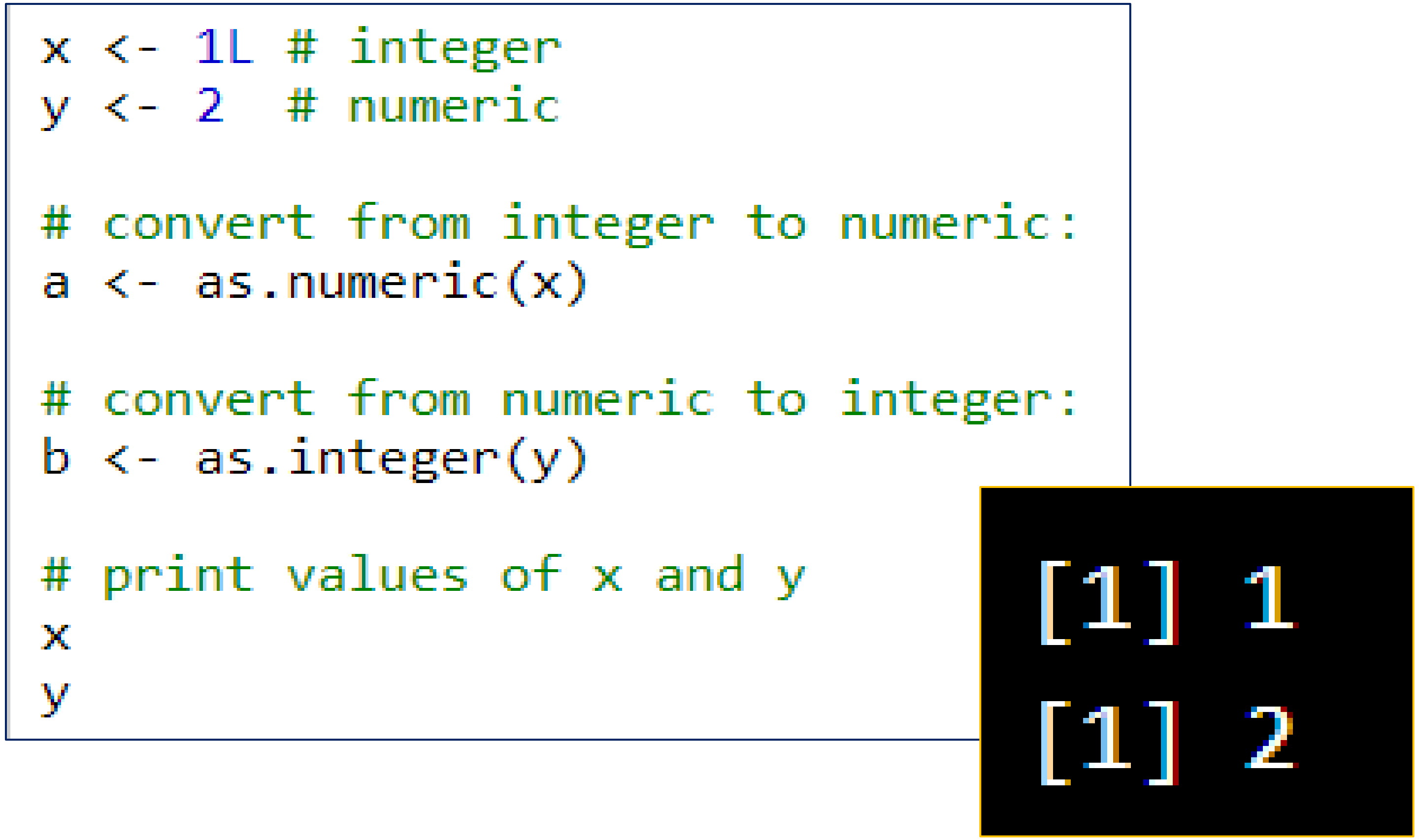


```
x <- 3+5i
y <- 5i

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

```
[1] 3+5i
[1] 0+5i
[1] "complex"
[1] "complex"
```

Type Conversion

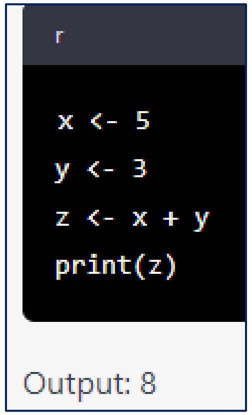
- as.numeric()
- as.integer()
- as.complex()



```
x <- 1L # integer
y <- 2  # numeric

# convert from integer to numeric:
a <- as.numeric(x)

# convert from numeric to integer:
b <- as.integer(y)

# print values of x and y
x
y
```

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

Arithmetic Operations



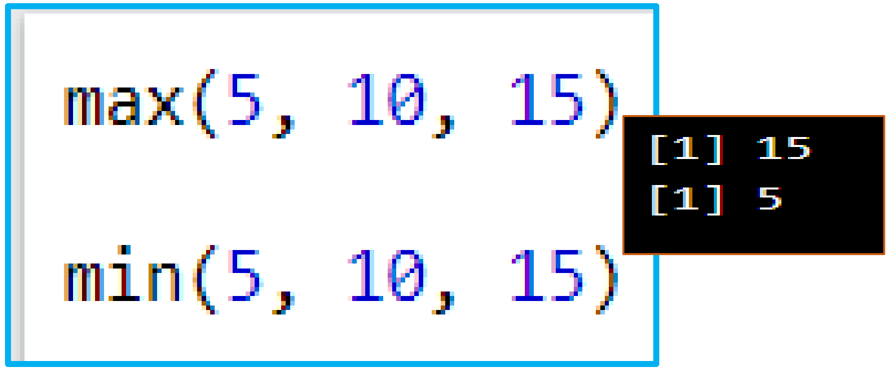
```r
r

x <- 5
y <- 3
z <- x + y
print(z)
```

Output: 8

- Addition: The addition operator in R is "+". For example:
- Subtraction: The subtraction operator in R is "-". For example:
- Multiplication: The multiplication operator in R is "*". For example:
- Division: The division operator in R is "/". For example:

Built-in Math Functions

- R also has many built-in math functions that allows you to perform mathematical tasks on numbers.
- For example, the min() and max() functions can be used to find the lowest or highest number in a set:



```
max(5, 10, 15)

min(5, 10, 15)
```

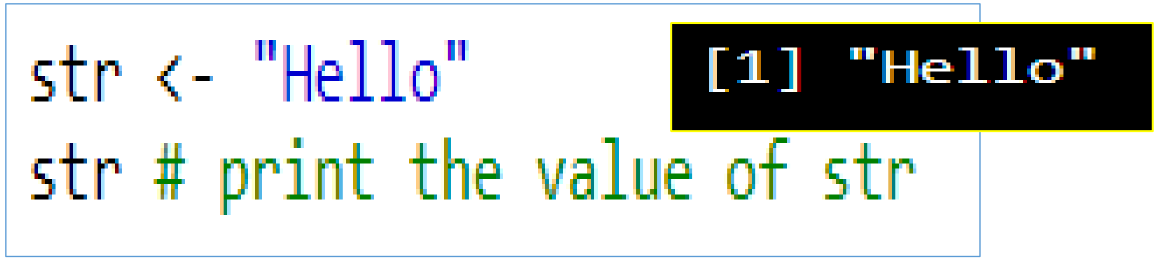
```
[1] 15
[1] 5
```

- abs() - This function returns the absolute value of a number. For example, abs(-5) would return 5.
- sqrt() - This function returns the square root of a number. For example, sqrt(25) would return 5.
- exp() - This function returns the exponential value of a number. For example, exp(2) would return 7.389056.
- log() - This function returns the natural logarithm of a number. For example, log(10) would return 2.302585.
- sin() - This function returns the sine of an angle in radians. For example, sin(pi/2) would return 1.

- cos() - This function returns the cosine of an angle in radians. For example, cos(pi) would return -1.
- tan() - This function returns the tangent of an angle in radians. For example, tan(pi/4) would return 1.
- min() and max() - These functions return the minimum and maximum values in a vector, respectively. For example, min(c(3, 6, 2, 9)) would return 2, while max(c(3, 6, 2, 9)) would return 9.
- sum() - This function returns the sum of all values in a vector. For example, sum(c(3, 6, 2, 9)) would return 20.
- mean() - This function returns the mean (average) of all values in a vector. For example, mean(c(3, 6, 2, 9)) would return 5.

String Literals

- A string is surrounded by either single quotation marks, or double quotation marks:

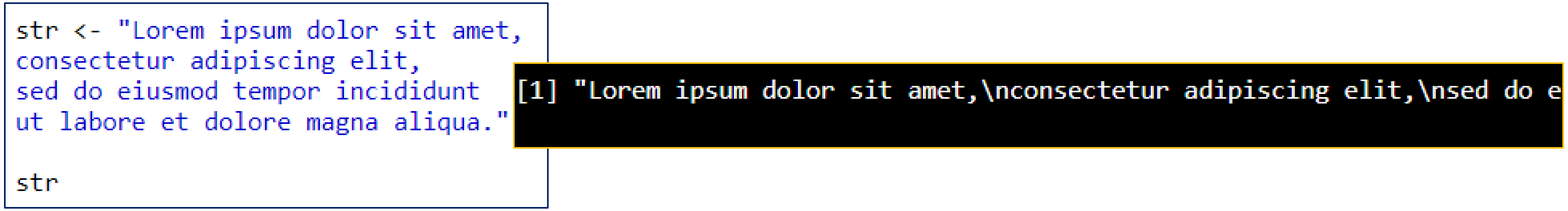


```
str <- "Hello"
str # print the value of str
```

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

- "hello" is the same as 'hello‘

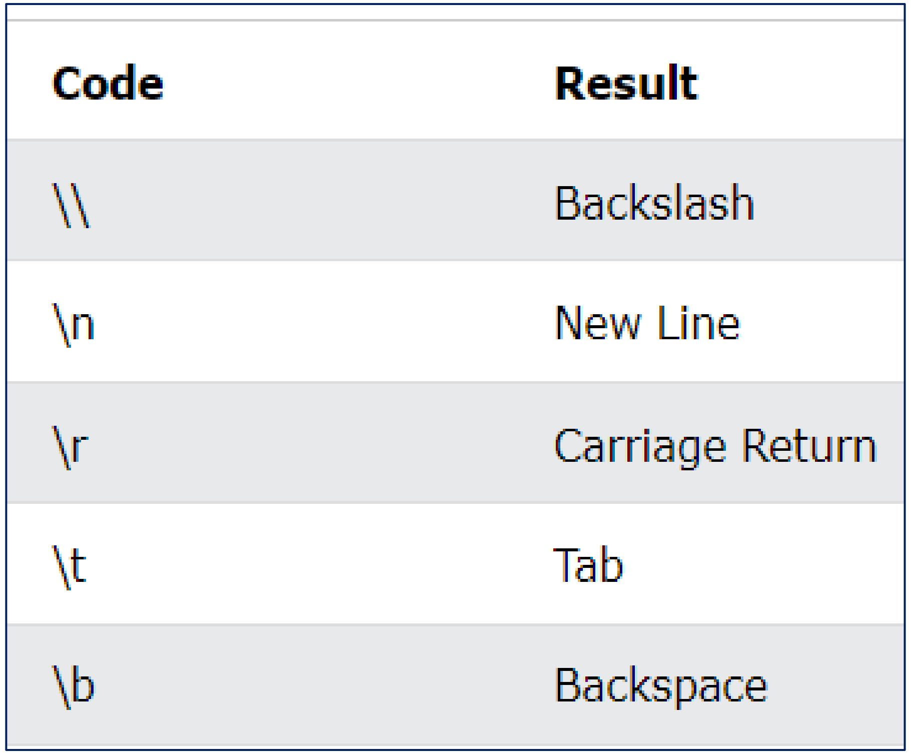
- You can assign a multiline string to a variable like this



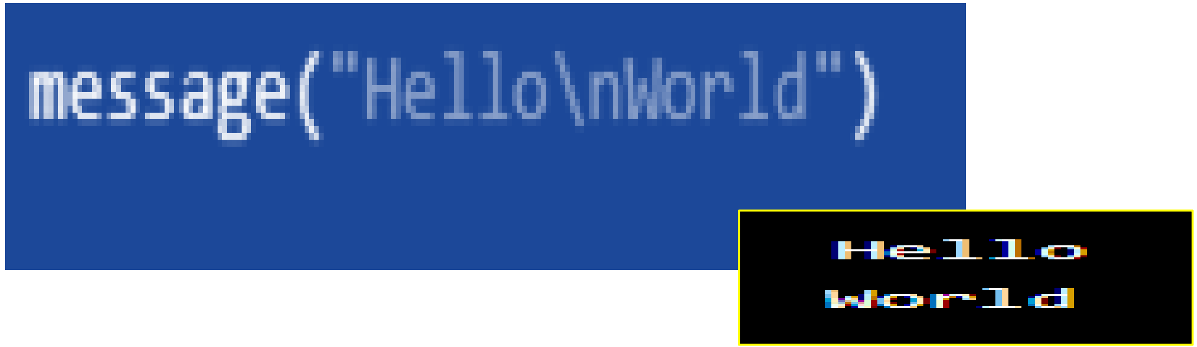
```
str <- "Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."

str
```

```
[1] "Lorem ipsum dolor sit amet,\nconsectetur adipiscing elit,\nsed do e
```

Escape Characters in R

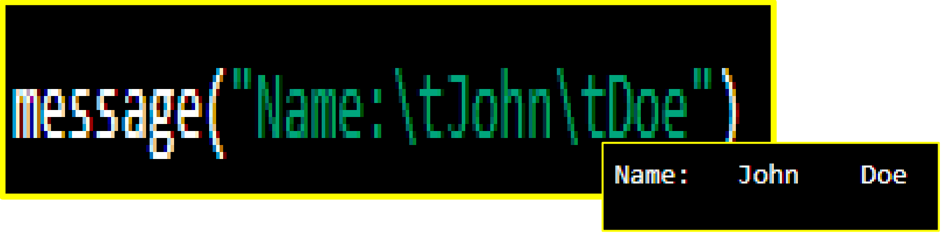


Code	Result
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace



```
message("Hello\nWorld")
```

```
Hello
World
```

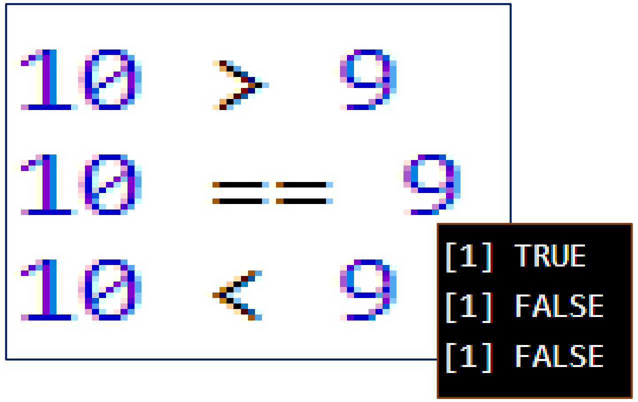


```
message("Name:\tJohn\tDoe")
```

```
Name:	John	Doe
```

Booleans
(Logical Values)

- In programming, you often need to know if an expression is true or false.
- You can evaluate any expression in R, and get one of two answers, TRUE or FALSE.
- When you compare two values, the expression is evaluated and R returns the logical answer:

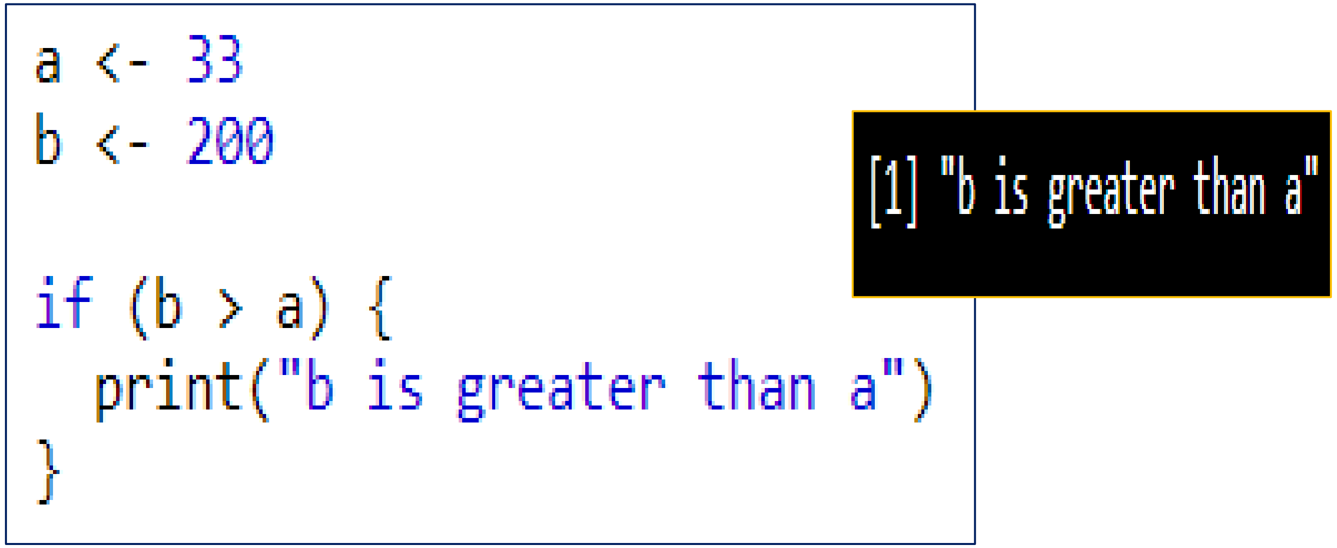


```
10 > 9
10 == 9
10 < 9
```

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

Conditions and If Statements

- The if Statement: An "if statement" is written with the if keyword, and it is used to specify a block of code to be executed if a condition is TRUE:

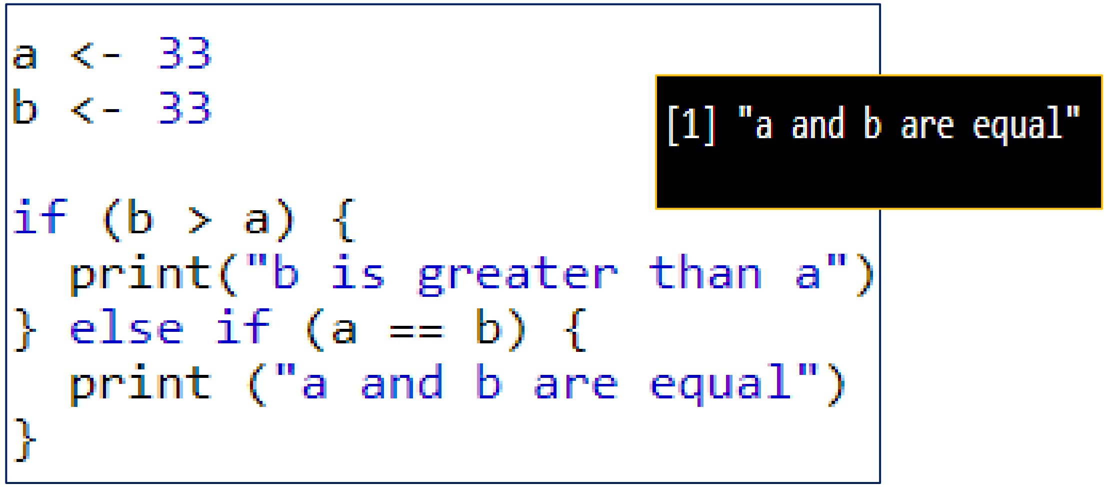


```r
a <- 33
b <- 200

if (b > a) {
  print("b is greater than a")
}
```

```
[1] "b is greater than a"
```

else if

- The else if keyword is R's way of saying "if the previous conditions were not true, then try this condition":

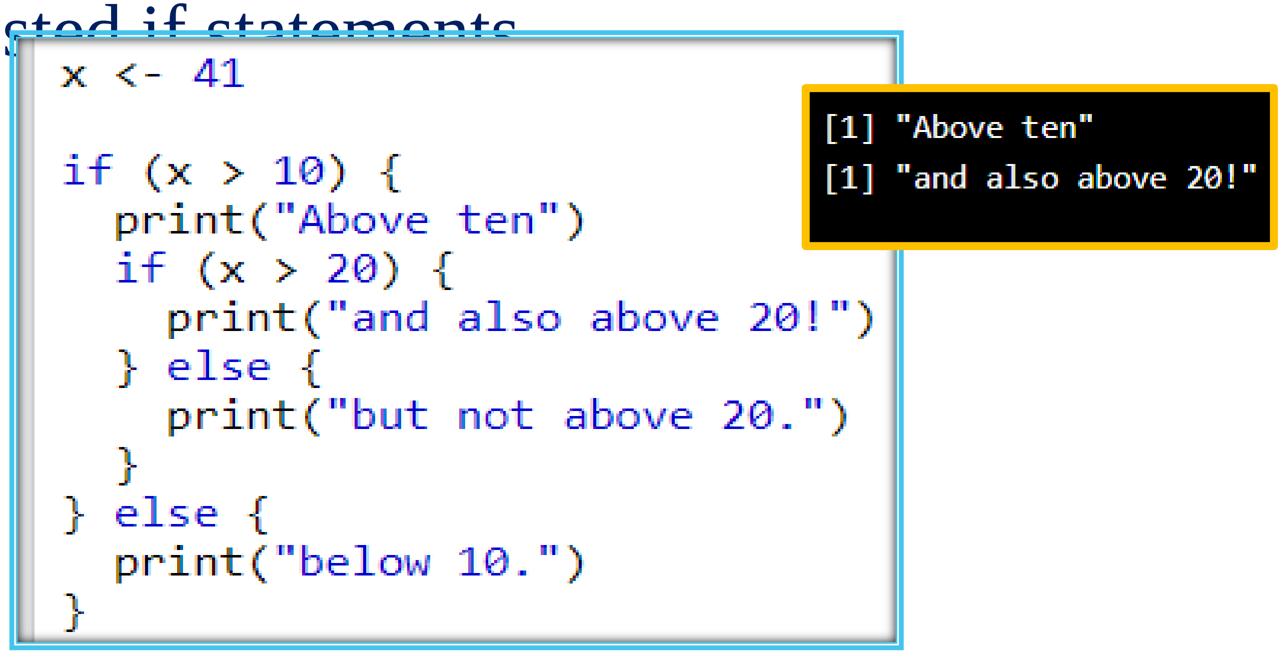


```
a <- 33
b <- 33

if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print ("a and b are equal")
}
```

```
[1] "a and b are equal"
```

Nested If Statements

- You can also have if statements inside if statements, this is called nested if statements.



```
x <- 41

if (x > 10) {
  print("Above ten")
  if (x > 20) {
    print("and also above 20!")
  } else {
    print("but not above 20.")
  }
} else {
  print("below 10.")
}
```

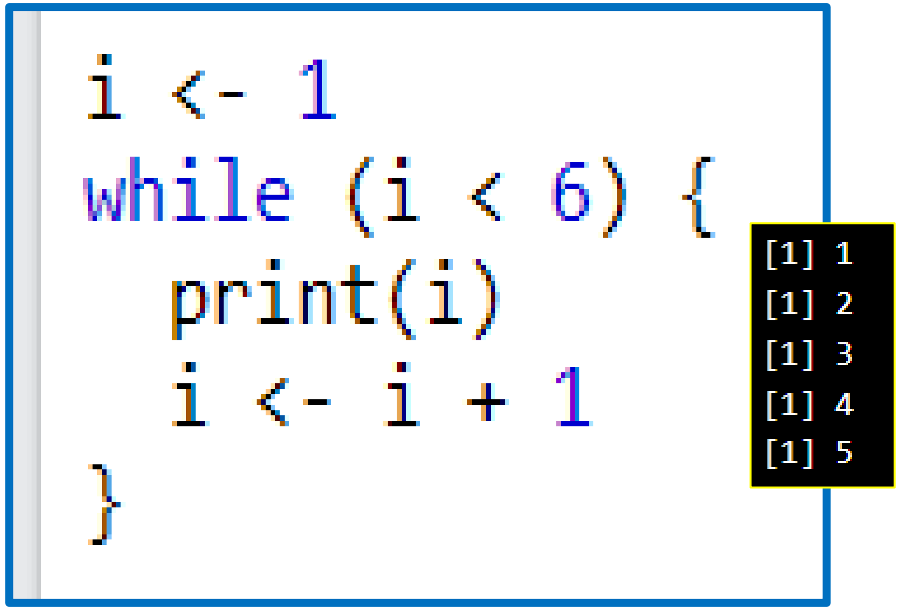
```
[1] "Above ten"
[1] "and also above 20!"
```

R While Loop

- Loops can execute a block of code as long as a specified condition is reached.
- Loops are handy because they save time, reduce errors, and they make code more readable.
- R has two loop commands:
- while loops
- for loops

while loop

- With the while loop we can execute a set of statements as long as a condition is TRUE:

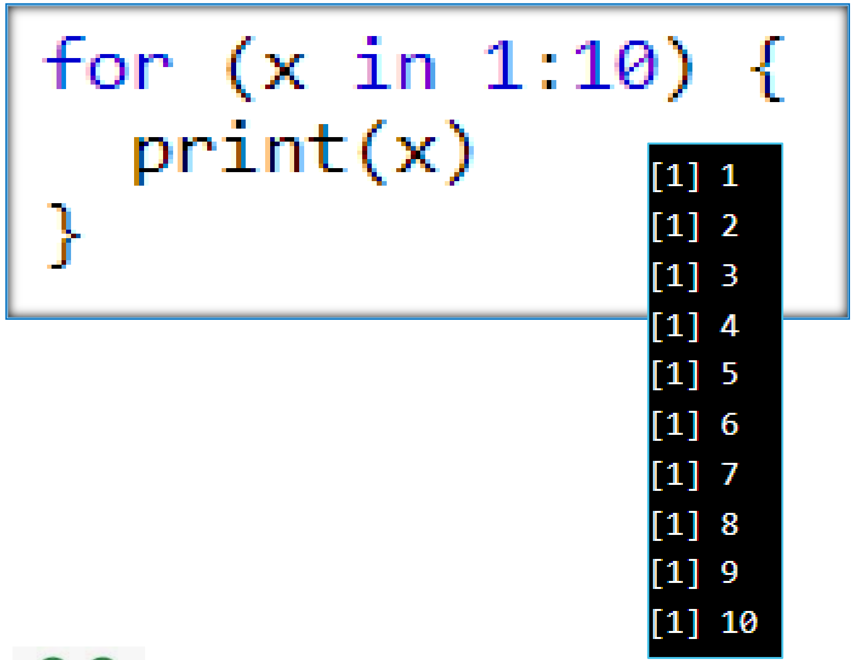


```
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

For loop

- A for loop is used for iterating over a sequence:



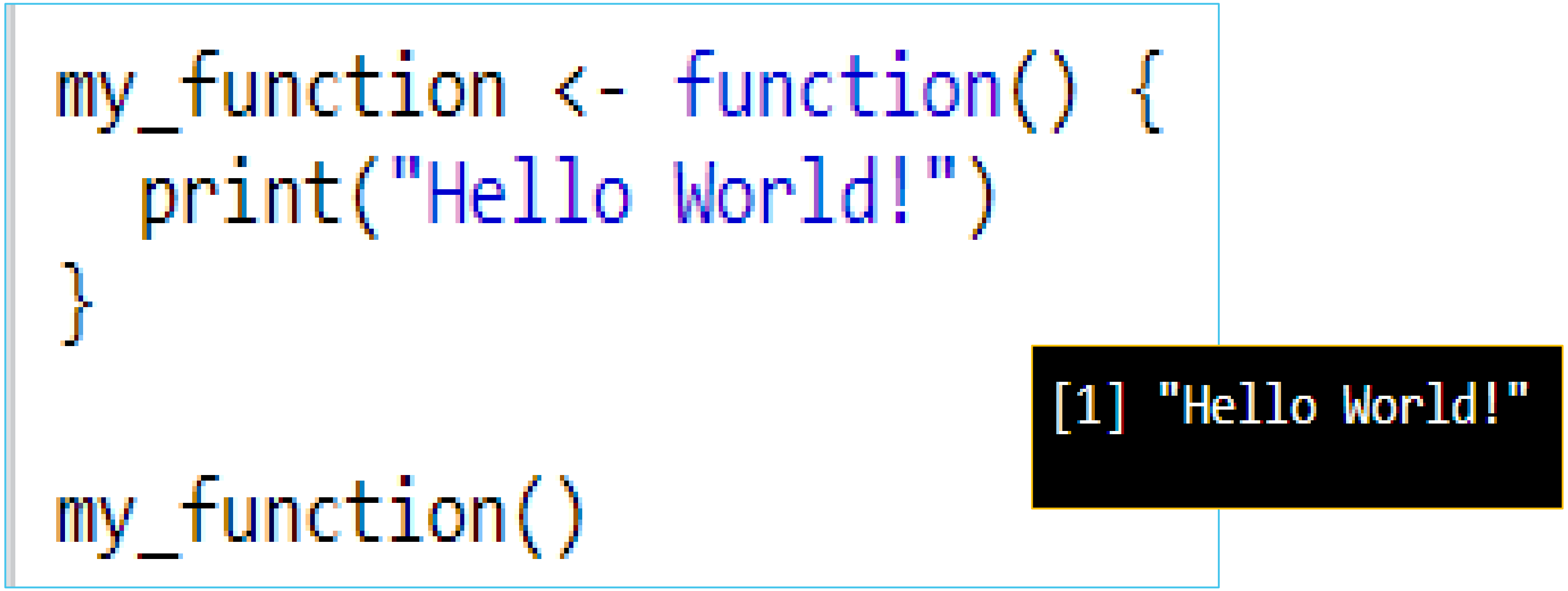
```
for (x in 1:10) {
  print(x)
}
```

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

R Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- To create a function, use the function() keyword:

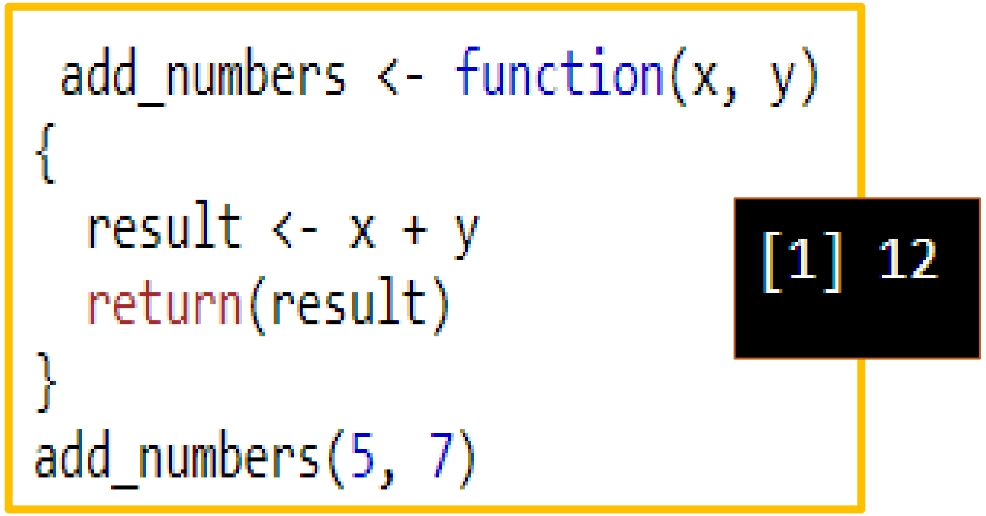
Example on user defined function



```
my_function <- function() {
  print("Hello World!")
}

my_function()
```

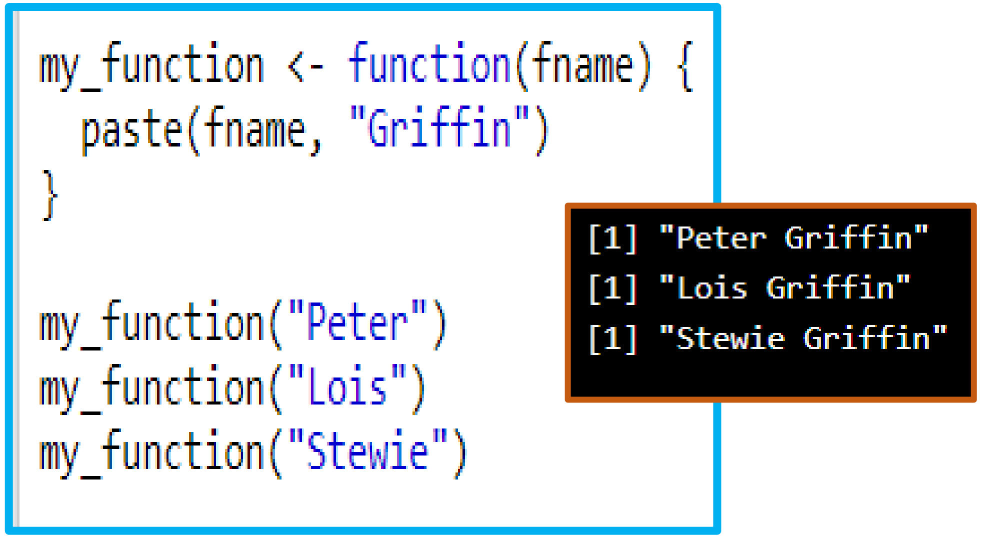
```
[1] "Hello World!"
```

User Defined functions Few Examples



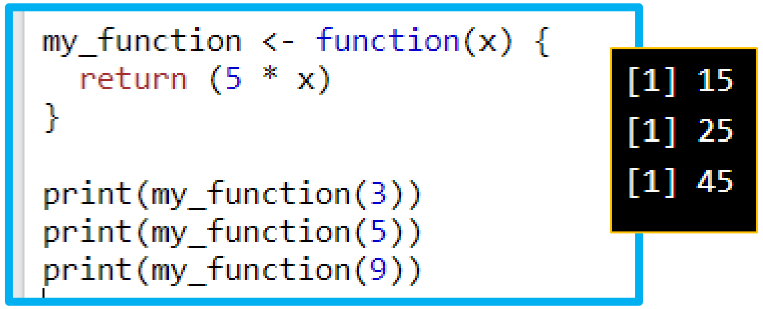
```
 add_numbers <- function(x, y)
{
  result <- x + y
  return(result)
}
add_numbers(5, 7)
```

```
[1] 12
```



```
my_function <- function(fname) {
  paste(fname, "Griffin")
}

my_function("Peter")
my_function("Lois")
my_function("Stewie")
```

```
[1] "Peter Griffin"
[1] "Lois Griffin"
[1] "Stewie Griffin"
```

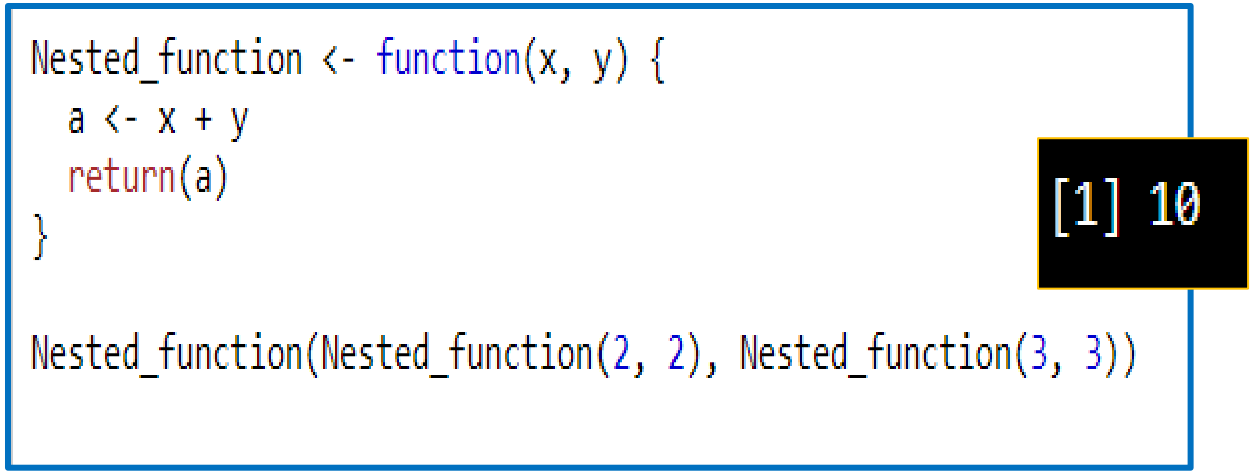


```
my_function <- function(x) {
  return (5 * x)
}

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

```
[1] 15
[1] 25
[1] 45
```

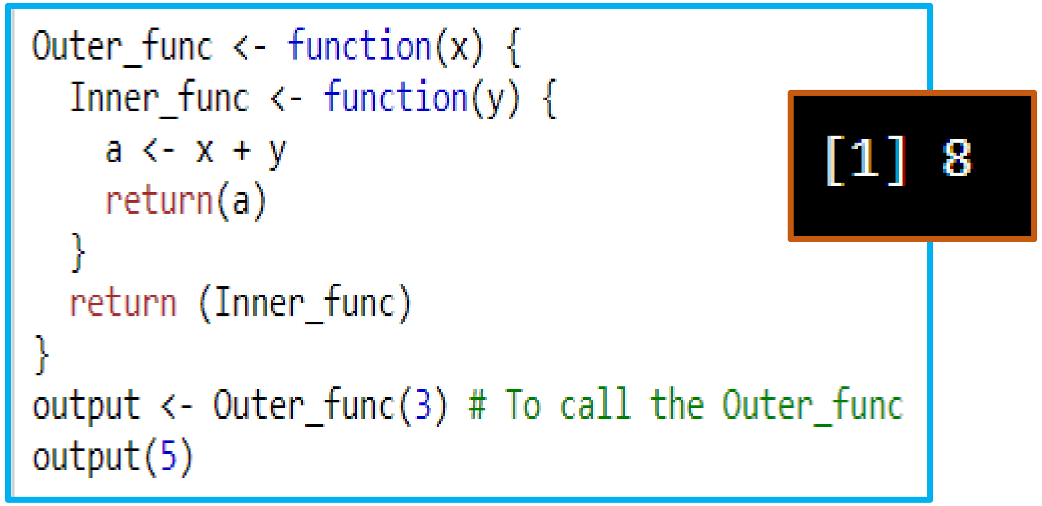
Nested Functions

- There are two ways to create a nested function:
- Call a function within another function.
- Write a function within a function.



```
Nested_function <- function(x, y) {
  a <- x + y
  return(a)
}

Nested_function(Nested_function(2, 2), Nested_function(3, 3))
```

```
[1] 10
```

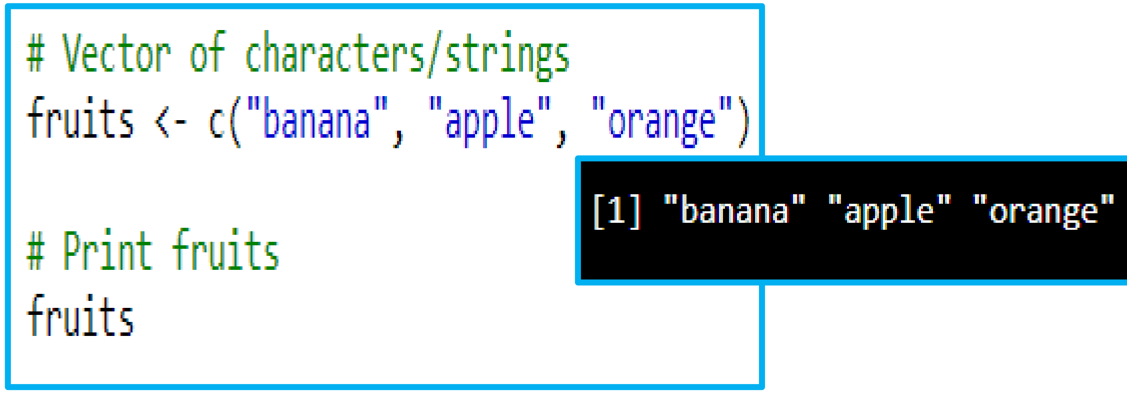


```
Outer_func <- function(x) {
  Inner_func <- function(y) {
    a <- x + y
    return(a)
  }
  return (Inner_func)
}
output <- Outer_func(3) # To call the Outer_func
output(5)
```

```
[1] 8
```

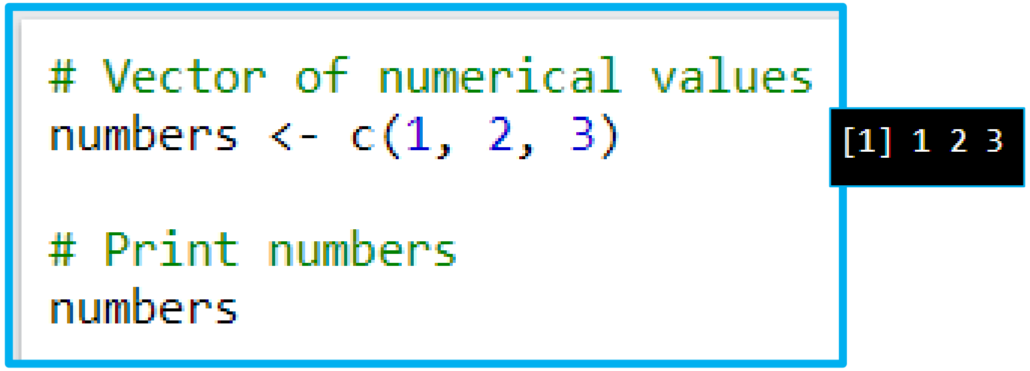
R Data Structures

- A vector is simply a list of items that are of the same type.
- To combine the list of items to a vector, use the c() function and separate the items by a comma.
- In the example below, we create a vector variable called fruits, that combine strings:



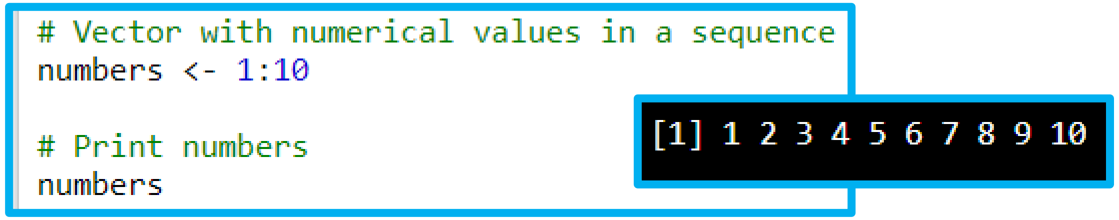
```
# Vector of characters/strings
fruits <- c("banana", "apple", "orange")

# Print fruits
fruits
```

```
[1] "banana" "apple" "orange"
```



```
# Vector of numerical values
numbers <- c(1, 2, 3)

# Print numbers
numbers
```

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

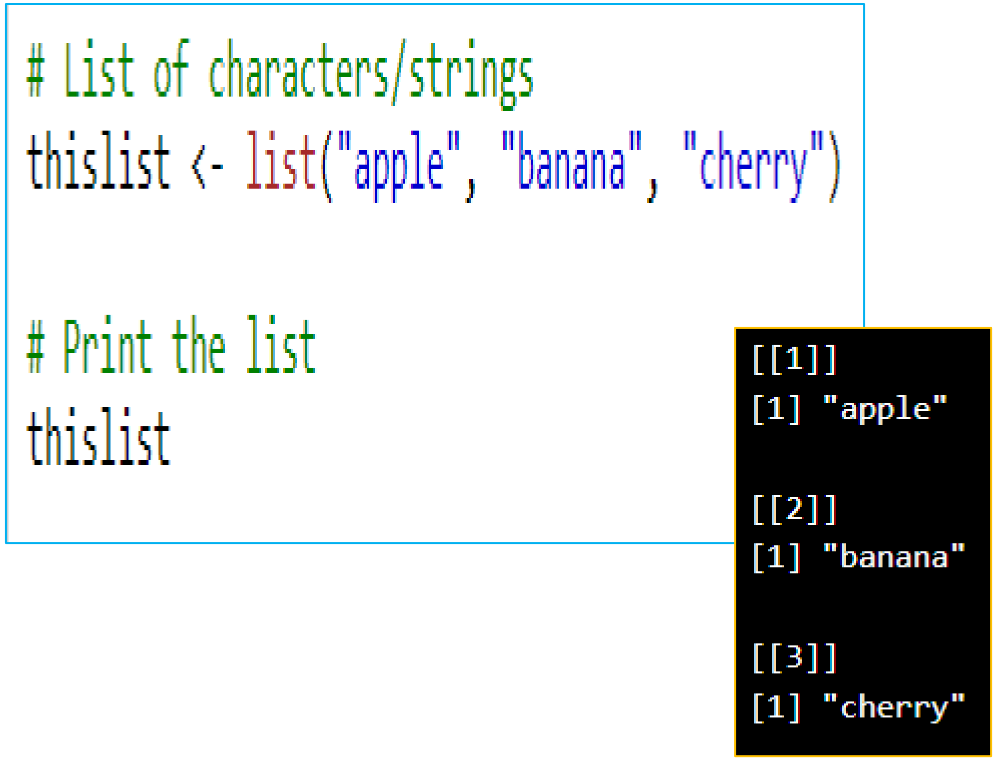


```
# Vector with numerical values in a sequence
numbers <- 1:10

# Print numbers
numbers
```

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

Lists

- A list in R can contain many different data types inside it. A list is a collection of data which is ordered and changeable.
- To create a list, use the list() function:

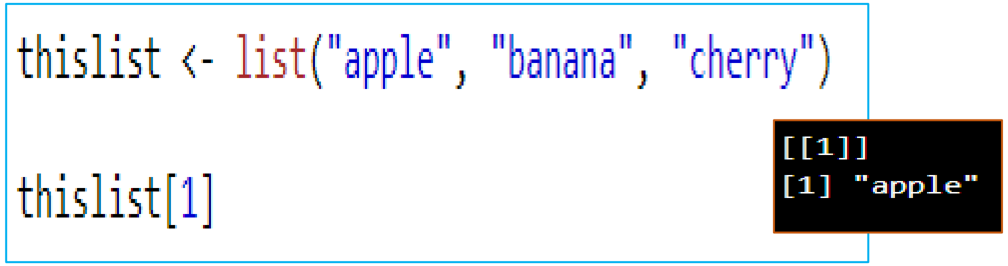


```
# List of characters/strings
thislist <- list("apple", "banana", "cherry")

# Print the list
thislist
```

```
[[1]]
[1] "apple"

[[2]]
[1] "banana"

[[3]]
[1] "cherry"
```

Accessing Lists

- You can access the list items by referring to its index number, inside brackets. The first item has index 1, the second item has index 2, and so on

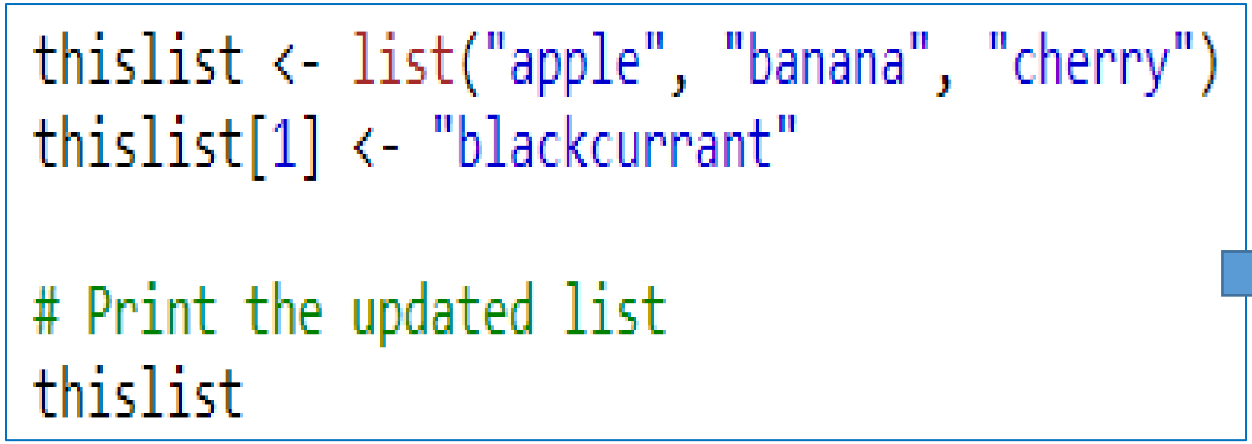


```
thislist <- list("apple", "banana", "cherry")

thislist[1]
```

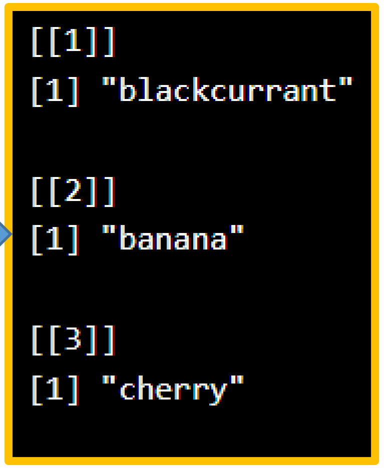
```
[[1]]
[1] "apple"
```

Change Item Value

- Change the value of a specific item, refer to the index number:



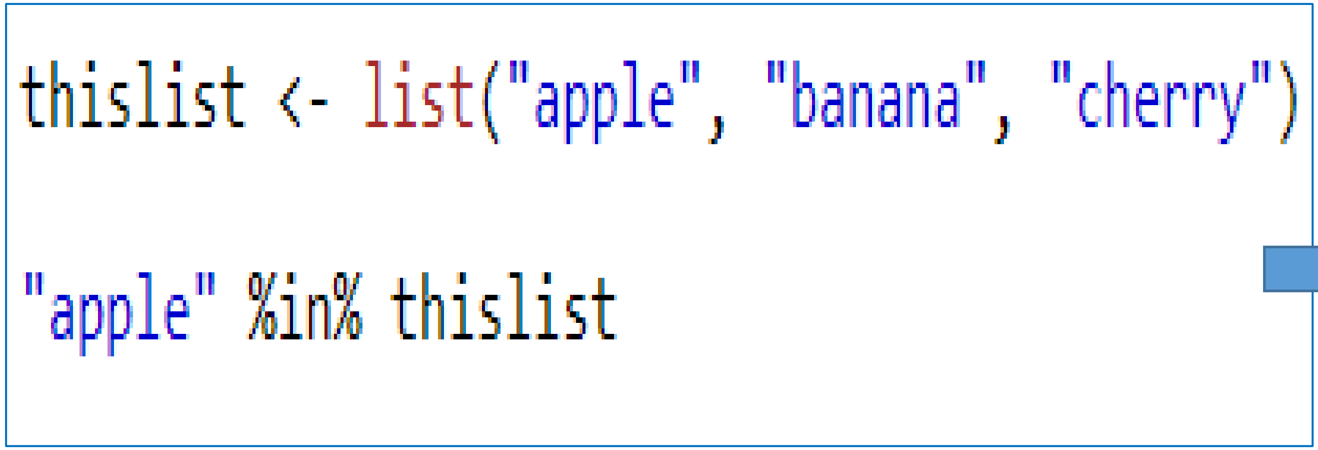
```r
thislist <- list("apple", "banana", "cherry")
thislist[1] <- "blackcurrant"

# Print the updated list
thislist
```



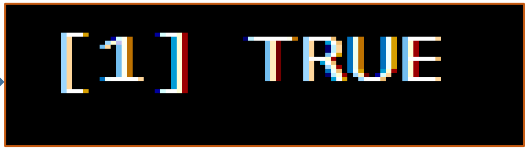
```
[[1]]
[1] "blackcurrant"

[[2]]
[1] "banana"

[[3]]
[1] "cherry"
```

Check if Item Exists

- To find out if a specified item is present in a list, use the %in% operator:



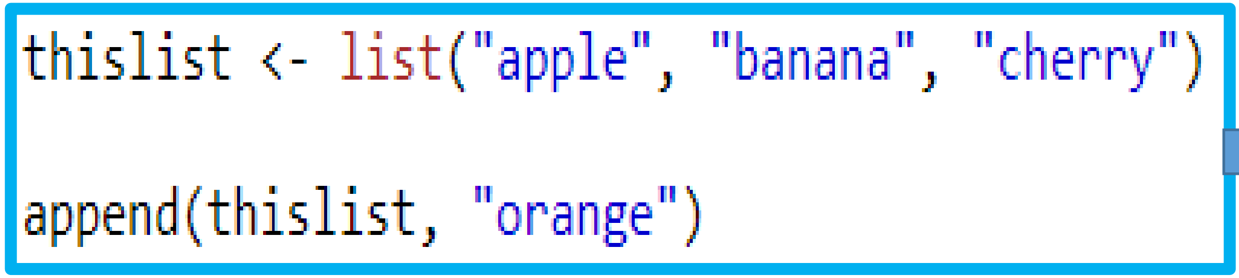
```
thislist <- list("apple", "banana", "cherry")

"apple" %in% thislist
```



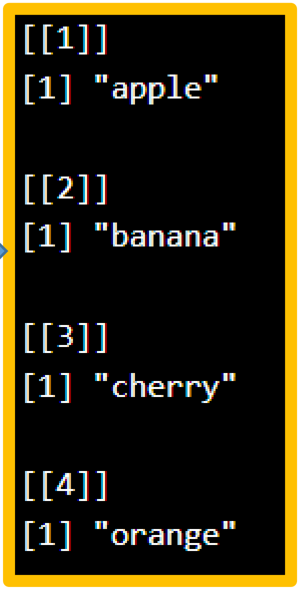
```
[1] TRUE
```

To Append the list

- To add an item to the end of the list, use the append() function:



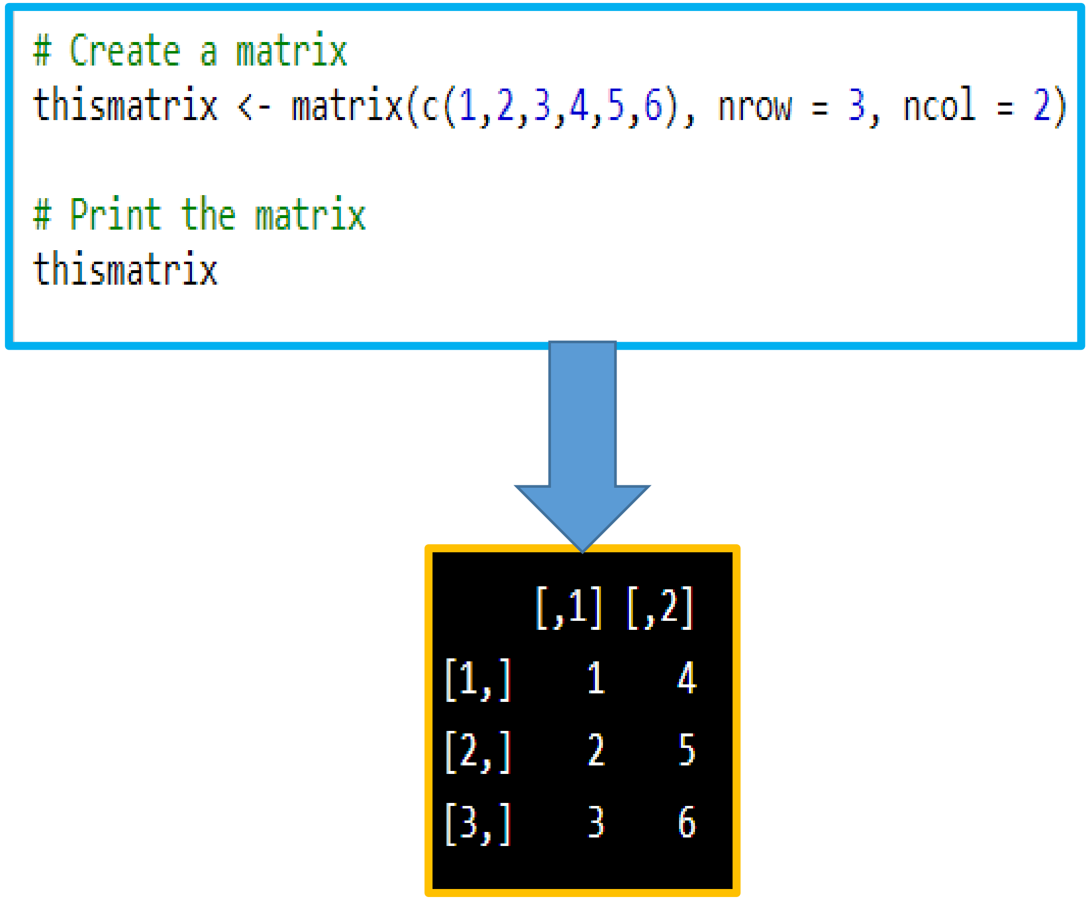
```
thislist <- list("apple", "banana", "cherry")

append(thislist, "orange")
```



```
[[1]]
[1] "apple"

[[2]]
[1] "banana"

[[3]]
[1] "cherry"

[[4]]
[1] "orange"
```

Reading and Writing Matrices in R

- A matrix is a two dimensional data set with columns and rows.
- A column is a vertical representation of data, while a row is a horizontal representation of data.
- A matrix can be created with the matrix() function. Specify the nrow and ncol parameters to get the amount of rows and columns:

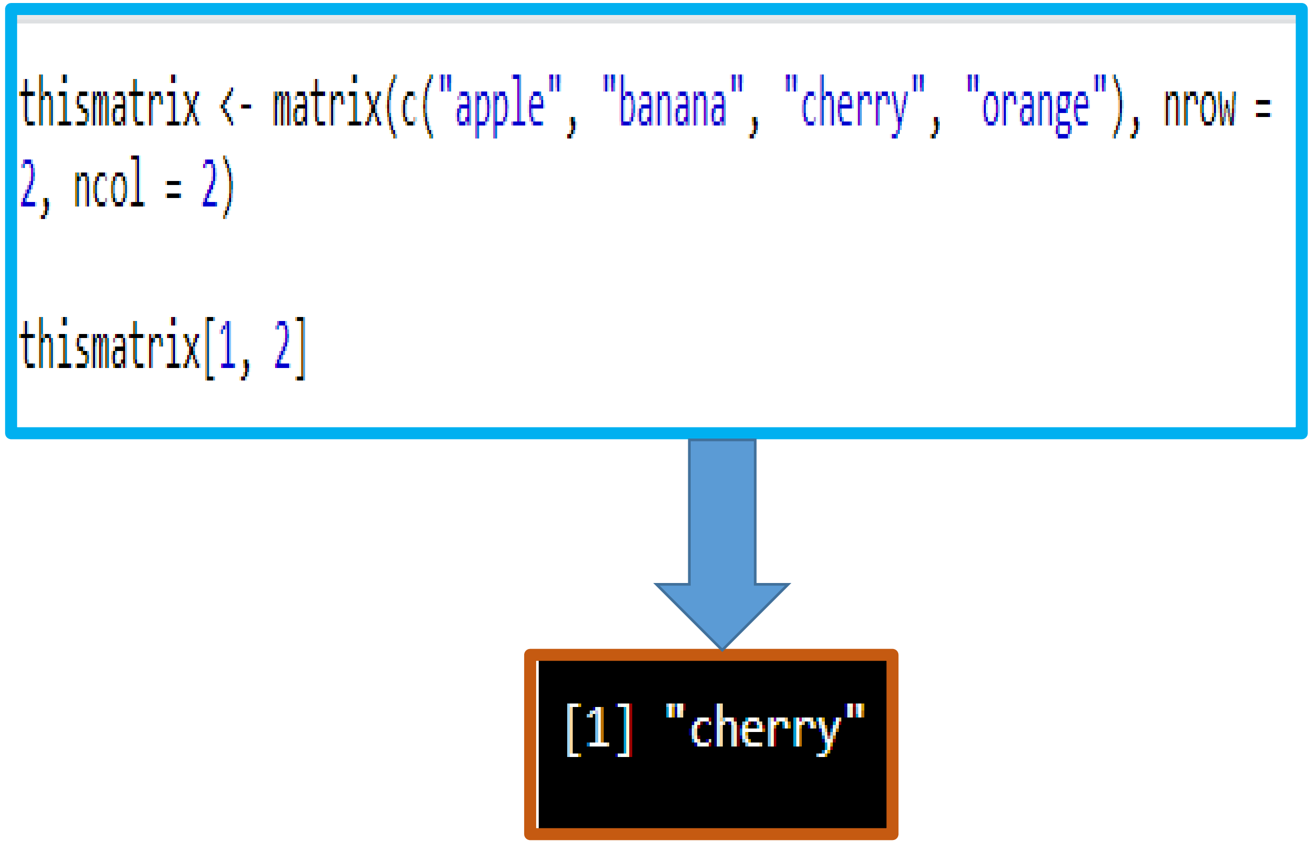


```
# Create a matrix
thismatrix <- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)

# Print the matrix
thismatrix
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Access Matrix Items

- You can access the items by using [ ] brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

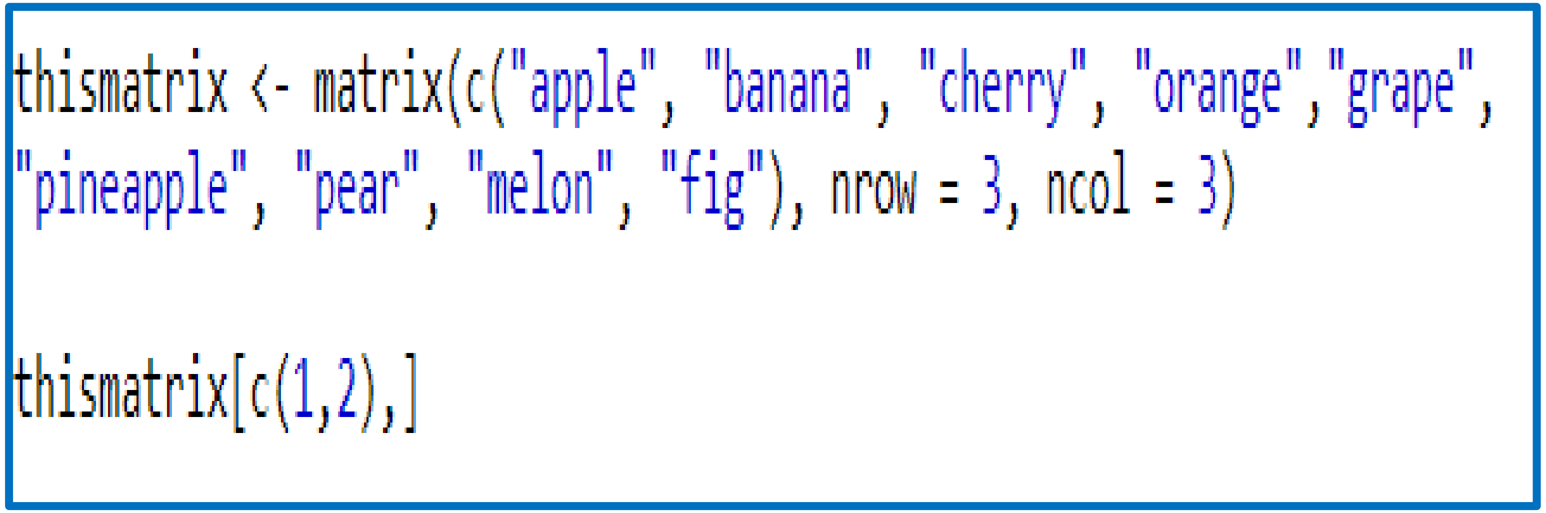


```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow =
2, ncol = 2)

thismatrix[1, 2]
```

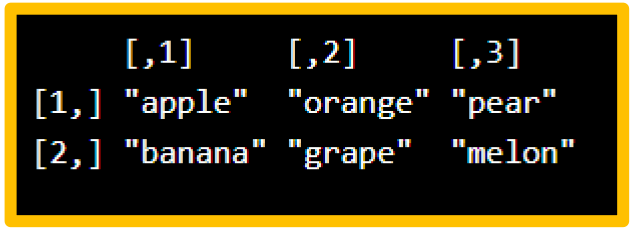
```
[1] "cherry"
```

Access Matrix Items

You can access the items by using [ ] brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

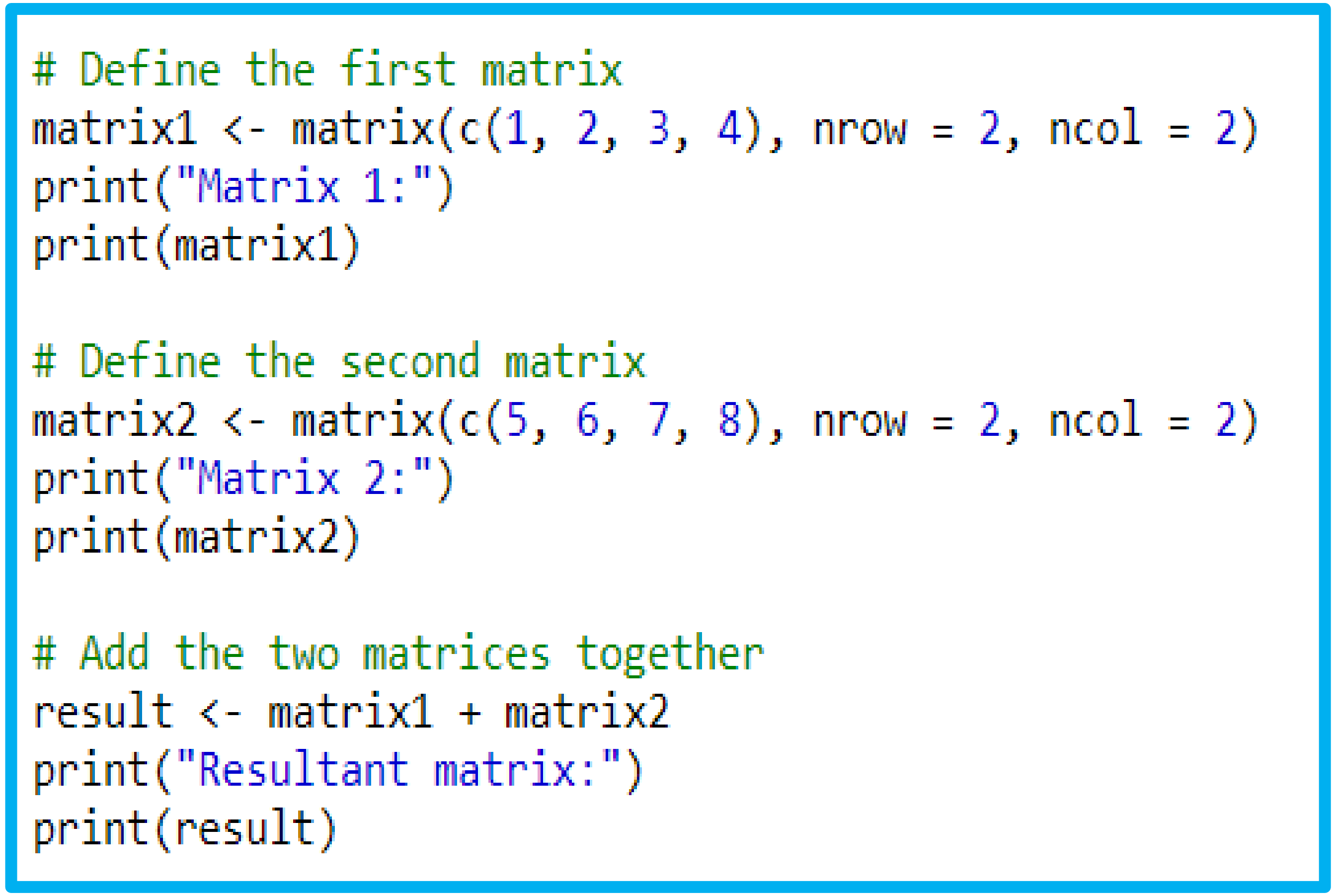


```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape",
"pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)

thismatrix[c(1,2),]
```

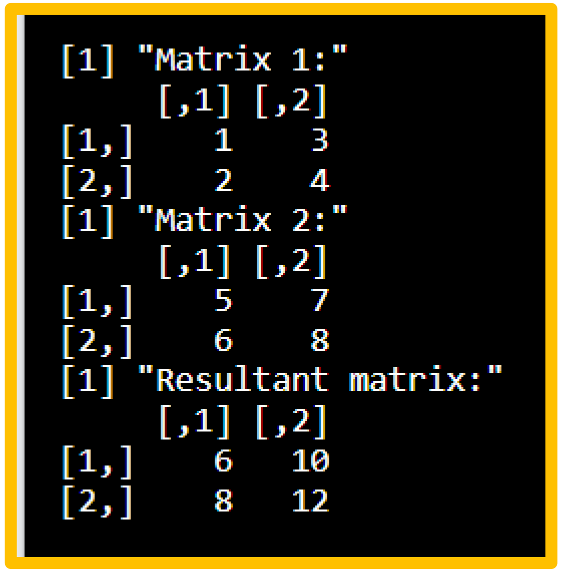


```
     [,1]     [,2]     [,3]
[1,] "apple"  "orange" "pear"
[2,] "banana" "grape"  "melon"
```

R Program to Add two Matrices

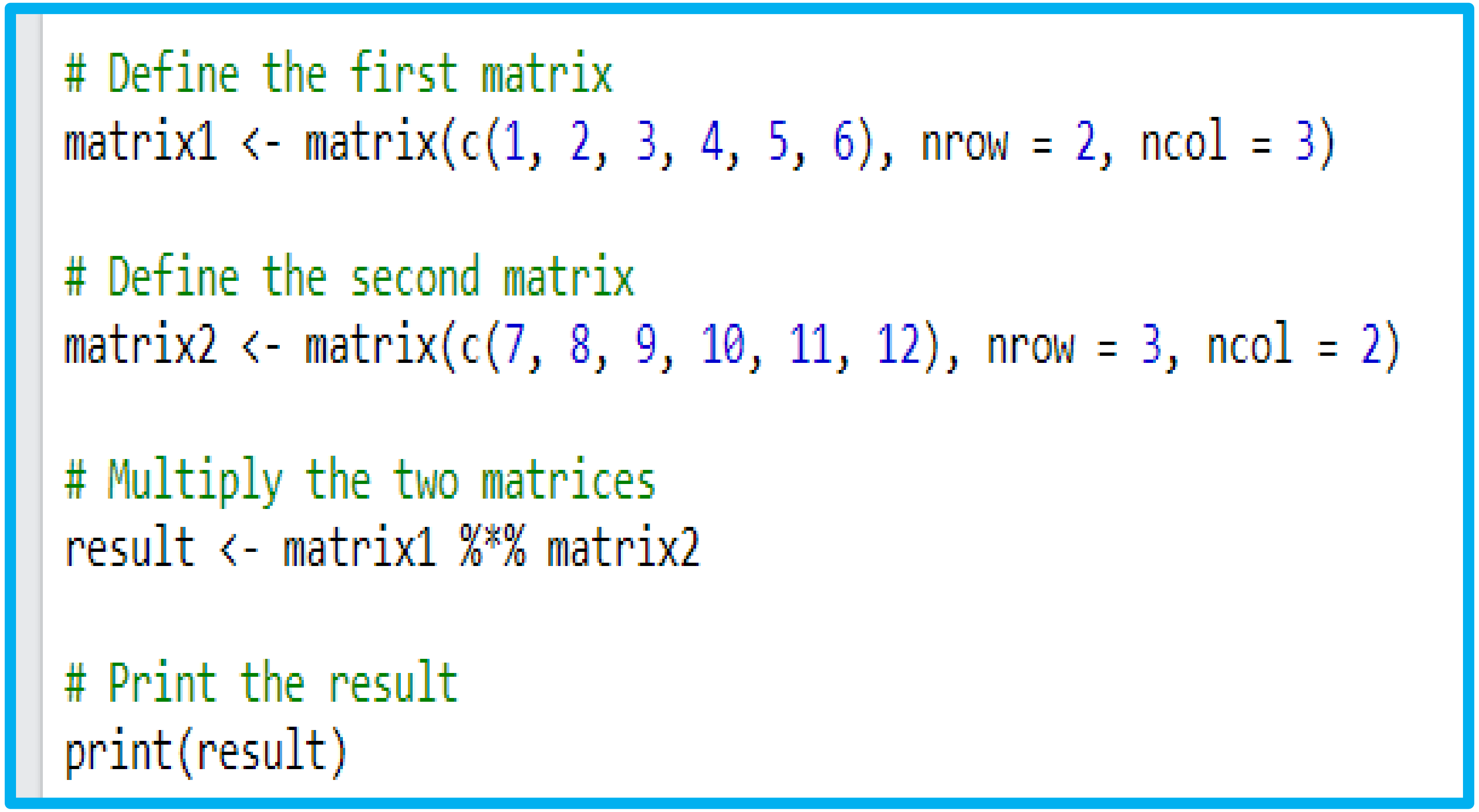


```r
# Define the first matrix
matrix1 <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
print("Matrix 1:")
print(matrix1)

# Define the second matrix
matrix2 <- matrix(c(5, 6, 7, 8), nrow = 2, ncol = 2)
print("Matrix 2:")
print(matrix2)

# Add the two matrices together
result <- matrix1 + matrix2
print("Resultant matrix:")
print(result)
```

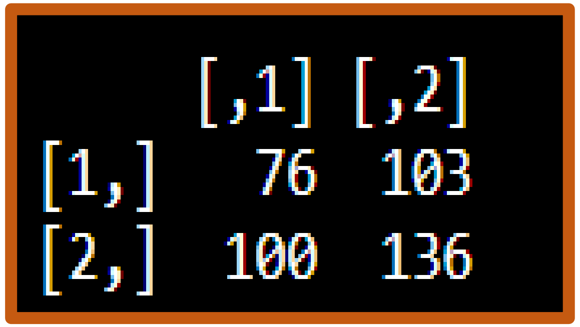


```
[1] "Matrix 1:"
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[1] "Matrix 2:"
     [,1] [,2]
[1,]    5    7
[2,]    6    8
[1] "Resultant matrix:"
     [,1] [,2]
[1,]    6   10
[2,]    8   12
```

R Program to Multiply Two Matrices



```
# Define the first matrix
matrix1 <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)

# Define the second matrix
matrix2 <- matrix(c(7, 8, 9, 10, 11, 12), nrow = 3, ncol = 2)

# Multiply the two matrices
result <- matrix1 %*% matrix2

# Print the result
print(result)
```



```
     [,1] [,2]
[1,]   76  103
[2,]  100  136
```