



# Syntax and Print in R

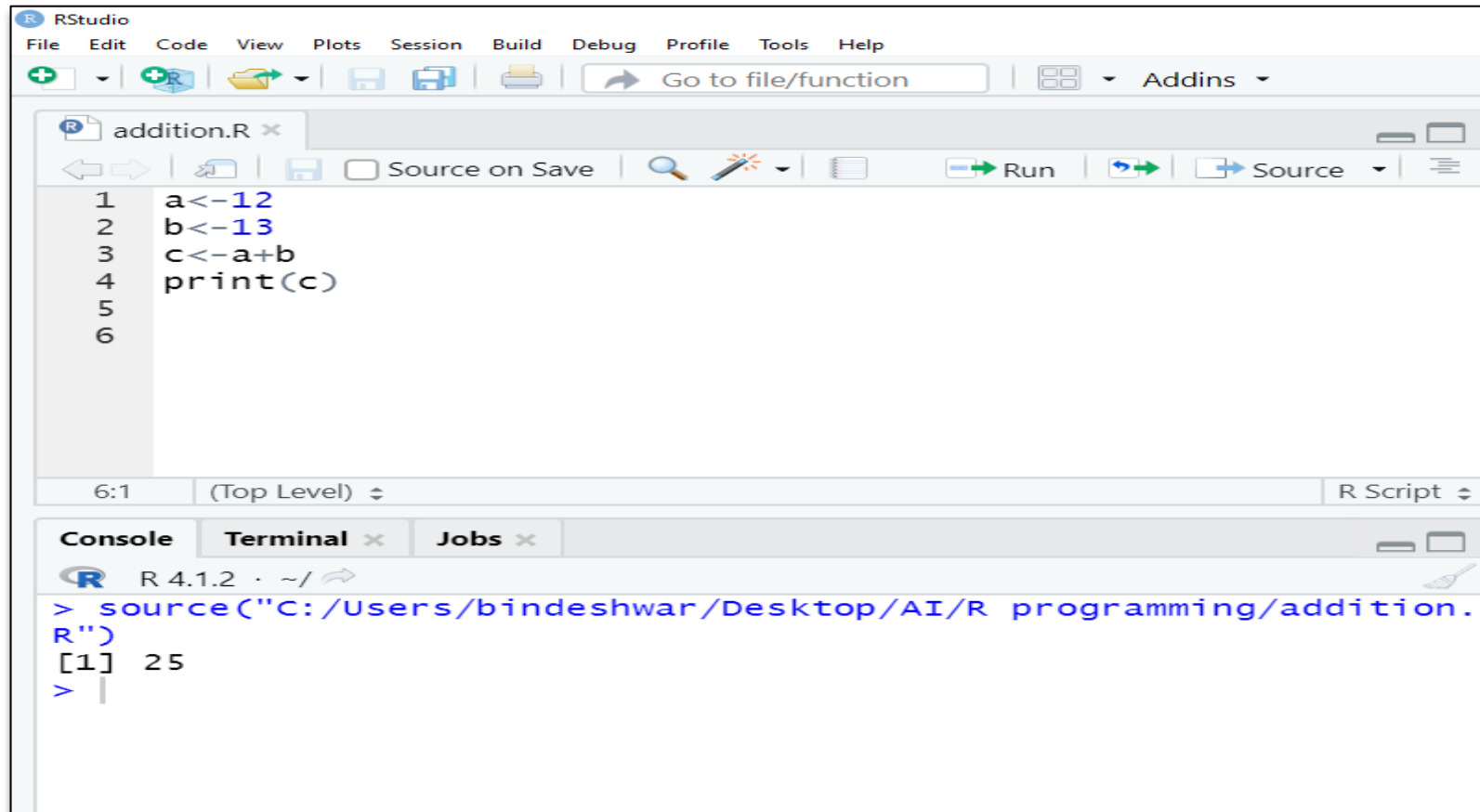
Part of Future Connect Media's IT Course

By Abdullah Hashmi



# Syntax in R

- The syntax in R refers to the rules and **structure** that dictate how you **write and format** R code. Proper syntax is crucial for the R interpreter to understand and execute your instructions correctly.



The screenshot displays the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The main editor window shows a script named 'addition.R' with the following code:

```
1 a<-12
2 b<-13
3 c<-a+b
4 print(c)
5
6
```

The status bar at the bottom of the editor indicates '6:1 (Top Level)' and 'R Script'. Below the editor is a console window with tabs for Console, Terminal, and Jobs. The console shows the execution of the script:

```
> source("C:/Users/bindeshwar/Desktop/AI/R programming/addition.R")
[1] 25
>
```

- Syntax: To output text in R, use single or double quotes:

Example: "Hello World!"

- To do simple calculations, add numbers together:

Example:

$5 + 5$



# Print in R

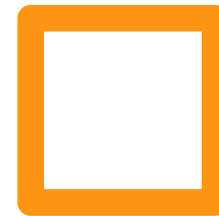
- Unlike many other programming languages, you can output code in R without using a print function:

Example:

```
"Hello World!"
```

- However, R does have a **print ()** function available if you want to use it.

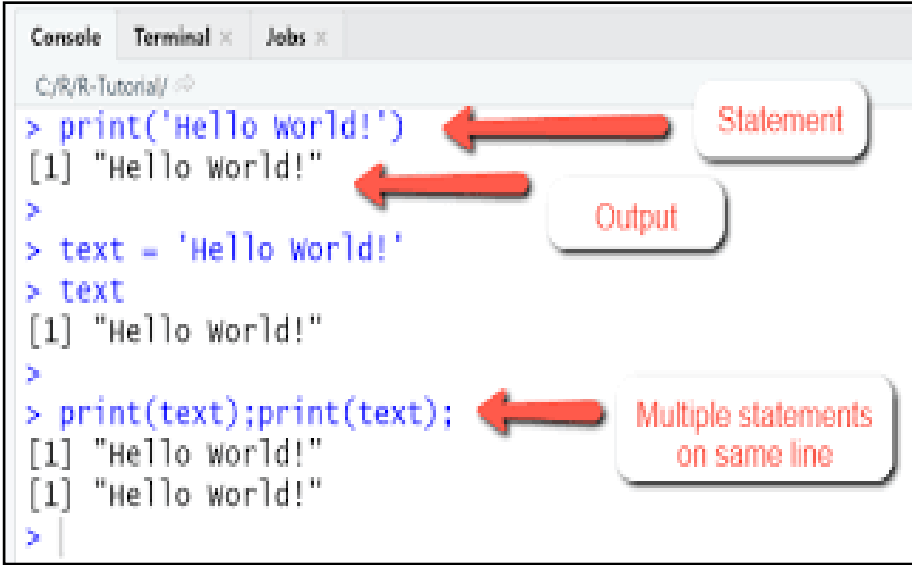
```
print ("Hello World!")
```



And there are times you must use the `print()` function to output code, for example when working with for loops (which you will learn more about in a later chapter):

### Example:

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



The screenshot shows an R console window with the following content:

```
C:/R/R-Tutorial/ > print('Hello world!')  
[1] "Hello World!"  
>  
> text = 'Hello world!'  
> text  
[1] "Hello World!"  
>  
> print(text);print(text);  
[1] "Hello World!"  
[1] "Hello World!"  
>
```

Annotations with red arrows point to specific parts of the code:

- Statement**: Points to the first line `> print('Hello world!')`.
- Output**: Points to the first output line `[1] "Hello World!"`.
- Multiple statements on same line**: Points to the line `> print(text);print(text);`.



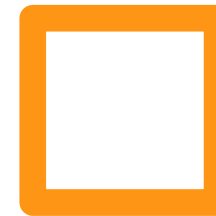
# Comments in R



Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

In R, comments are used to add **explanatory** notes or **documentation** within your code. Comments are ignored by the R interpreter when it runs your code, making them a useful tool for providing **context** and **explanations** to yourself and others who read your code.



# There are two common ways to add comments in R:

1. **\*\*Single-Line Comments\*\***: These comments are used for brief explanations within a single line of code. They start with the `#` symbol and continue until the end of the line. Here's an example:

```
""R
```

```
# This is a single-line comment
```

```
x <- 10 # Assigning a value to the variable x
```

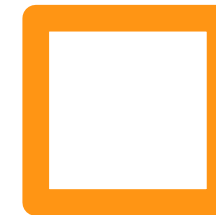
```
""
```



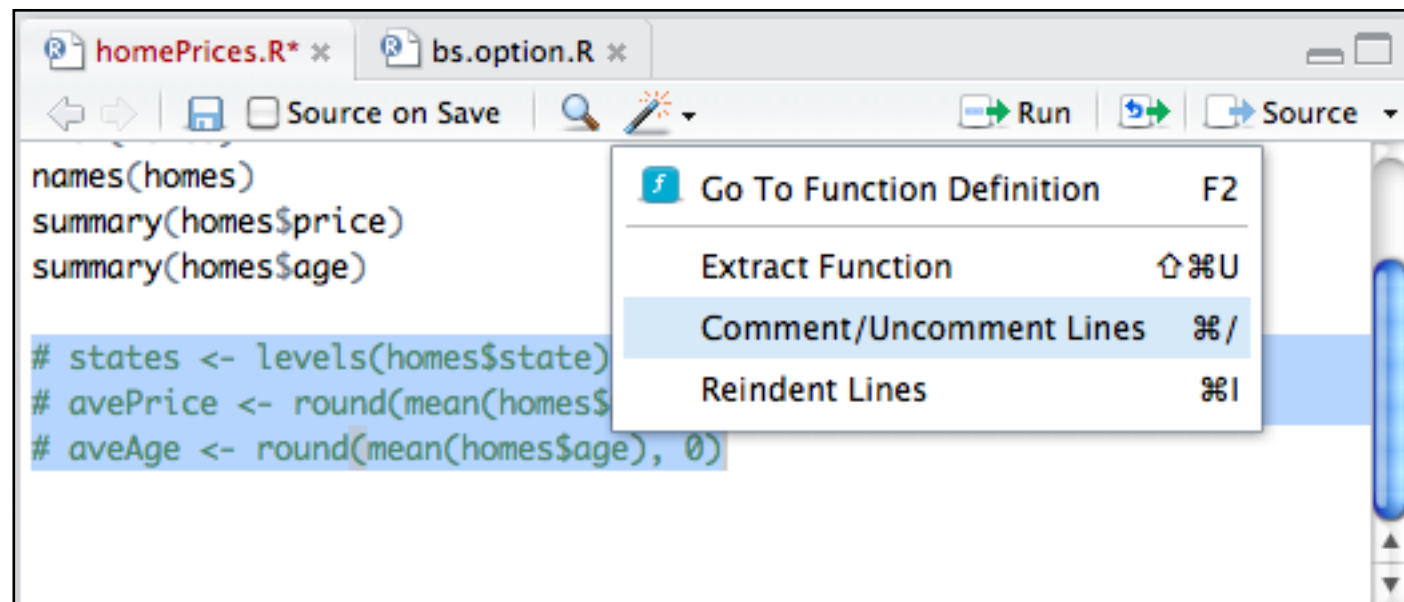
2. **\*\*Multi-Line Comments\*\***: Although R doesn't have a dedicated syntax for multi-line comments like some other programming languages, you can create multi-line comments by using the `#` symbol at the beginning of each line.

**Here's an example:**

```
""R
# This is a multi-line comment.
# You can add as many lines as you need.
# It's often used for longer explanations.
""
```



- Comments does not have to be text to explain the code, it can also be used to prevent R from executing the code.
- Unlike other programming languages, such as Java, there are no syntax in R for multiline comments. However, we can just insert a # for each line to create multiline comments:



The screenshot shows the RStudio IDE with two open files: `homePrices.R*` and `bs.option.R`. The `homePrices.R` file contains the following code:

```
names(homes)
summary(homes$price)
summary(homes$age)
# states <- levels(homes$state)
# avePrice <- round(mean(homes$
# aveAge <- round(mean(homes$age), 0)
```

A context menu is open over the commented-out lines, showing the following options:

- Go To Function Definition (F2)
- Extract Function (⇧⌘U)
- Comment/Uncomment Lines (⌘/)
- Reindent Lines (⌘I)



# Variables in R



Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

# Creating Variables in R

- 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:

## Example:

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

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



- From the example above, name and age are variables, while "John" and 40 are values.
- In other programming language, it is common to use = as an assignment operator. In R, we can use both = and <- as assignment operators.
- However, <- is preferred in most cases because the = operator can be forbidden in some context in R.
- Print / Output Variables
- Compared to many other programming languages, you do not have to use a function to print/output variables in R. You can just type the name of the variable:

## Example:

```
`name <- "John Doe"
      name # auto-print the value of
the name variable
```

- However, R does have a `print()` function available if you want to use it. This might be useful if you are familiar with other programming languages, such as Python, which often use a `print()` function to output variables.



# Concatenate Elements in R

- Concatenate Elements:
- You can also concatenate, or join, two or more elements, by using the `paste()` function.
- To combine both text and a variable, R uses comma (,):

## Example:

```
text <- "awesome"  
paste ("R is", text)
```

You can also use `,` to add a variable to another variable:

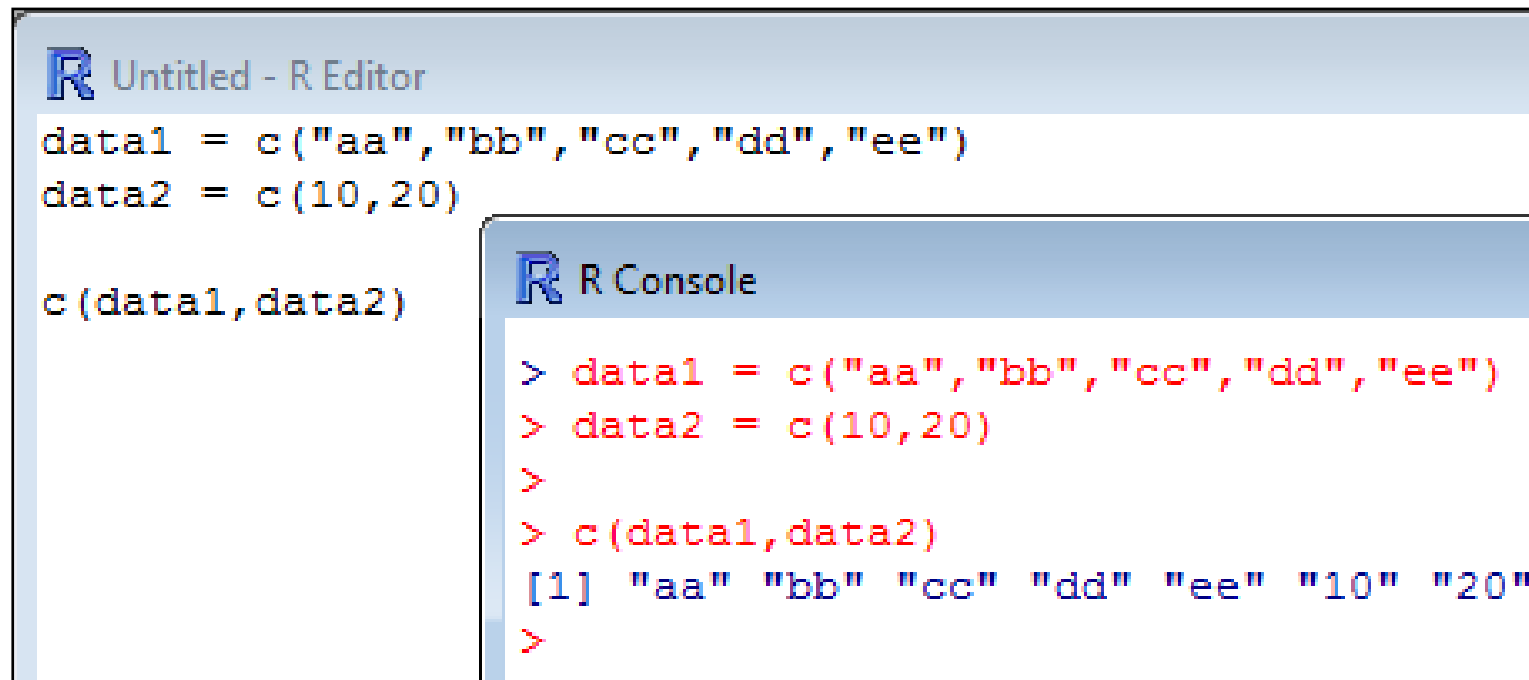
## Example:

```
text1 <- "R is"  
text2 <- "awesome"  
paste(text1, text2)
```

- For numbers, the + character works as a mathematical operator:

**Example:**

```
num1 <- 5  
num2 <- 10  
num1 + num2
```



The image shows a screenshot of the R environment. On the left is the 'R Untitled - R Editor' window, and on the right is the 'R Console' window. The editor contains the following code:

```
data1 = c("aa","bb","cc","dd","ee")  
data2 = c(10,20)  
  
c(data1,data2)
```

The console shows the execution of these commands:

```
> data1 = c("aa","bb","cc","dd","ee")  
> data2 = c(10,20)  
>  
> c(data1,data2)  
[1] "aa" "bb" "cc" "dd" "ee" "10" "20"  
>
```



# Multiple Variables:

R allows you to assign the same value to multiple variables in one line:

## Example:

```
# Assign the same value to multiple variables in one line  
var1 <- var2 <- var3 <- "Orange"
```

```
# Print variable values  
var1  
var2  
var3
```

# Variable Names( Identifiers):

A variable can have a short name (**like x and y**) or a more descriptive name (**age, carname, total\_volume**). Rules for R variables are:

A variable name must start with a letter and can be a combination of letters, digits, period(.) and underscore(\_). If it starts with period(.), it cannot be followed by a digit.

A variable name cannot start with a number or underscore (\_)

Variable names are case-sensitive (age, Age and AGE are three different variables)

Reserved words cannot be used as variables (**TRUE, FALSE, NULL, if...**)

```
# Legal variable names:
```

```
myvar <- "John"
```

```
my_var <- "John"
```

```
myVar <- "John"
```

```
MYVAR <- "John"
```

```
myvar2 <- "John"
```

```
.myvar <- "John"
```

```
# Illegal variable names:
```

```
2myvar <- "John"
```

```
my-var <- "John"
```

```
my var <- "John"
```

```
_my_var <- "John"
```

```
my_v@ar <- "John"
```

```
TRUE <- "John"
```

- Remember that variable names are case-sensitive!



# Data Types in R

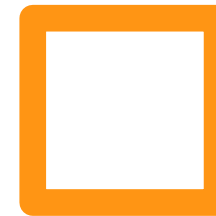


Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

# What is data type?

In programming, a data type defines the kind of value a variable can hold, and it determines what operations can be performed on that value. Common data types include integers (whole numbers), floats (numbers with decimals), strings (text), booleans (true/false), lists (ordered collections), and more. Data types help ensure proper storage and manipulation of data in a program.



## Data types in R:

In R programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

In R, variables do not need to be declared with any particular type, and can even change type after they have been set:

### Example:

```
my_var <- 30 # my_var is type of numeric
```

```
my_var <- "Sally" # my_var is now of type character (aka string)
```

R has a variety of data types and object classes. You will learn much more about these as you continue to get to know R.

## Basic Data Types:

Basic data types in R can be divided into the following types:

numeric - (10.5, 55, 787)

integer - (1L, 55L, 100L, where the letter "L" declares this as an integer)

complex - ( $9 + 3i$ , where "i" is the imaginary part)

character (a.k.a. string) - ("k", "R is exciting", "FALSE", "11.5")

logical (a.k.a. boolean) - (TRUE or FALSE)

## We can use the `class()` function to check the data type of a variable:

### Example:

```
# numeric  
x <- 10.5  
class(x)
```

```
# integer  
x <- 1000L  
class(x)
```

```
# complex  
x <- 9i + 3  
class(x)
```

```
# character/string  
x <- "R is exciting"  
class(x)
```

```
# logical/boolean  
x <- TRUE  
class(x)
```





# R Numbers



Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

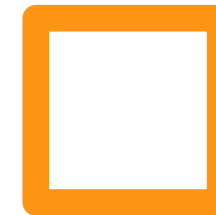
# There are three number types in R:

- **numeric**
- **integer**
- **complex**

Variables of number types are created when you assign a value to them:

## Example:

```
x <- 10.5    # numeric  
y <- 10L     # integer  
z <- 1i      # 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:

## Example:

```
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)
```

# 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:

## Example:

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

# Print values of x and y
x
y

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

# Complex:

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

## Example:

```
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)
```

# Type Conversion:

You can convert from one type to another with the following functions:

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

## Example:

```
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

# print the class name of a and b
class(a)
class(b)
```



# Math with R



Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

# Simple Math:

- In R, you can use operators to perform common mathematical operations on numbers.
- The  $+$  operator is used to add together two values:

**Example:  $10 + 5$ :**

- And the  $-$  operator is used for subtraction:

**Example:  $10 - 5$**





# 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:

## Example:

```
max(5, 10, 15)
```

```
min(5, 10, 15)
```

## sqrt():

The sqrt() function returns the square root of a number:

**Example:** sqrt(16)

# abs():

The abs() function returns the absolute **(positive)** value of a number:

**Example:** abs(-4.7)

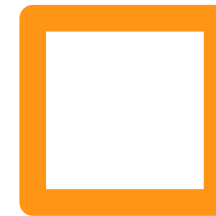
## ceiling() and floor():

The **ceiling()** function rounds a number upwards to its nearest integer, and the **floor()** function rounds a number downwards to its nearest integer, and returns the result:

**Example:**

ceiling(1.4)

floor(1.4)





# R Strings



Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

# String Literals:

- Strings are used for storing text.
- A string is surrounded by either single quotation marks, or double quotation marks:
- "hello" is the same as 'hello':

## Example:

"hello"

'hello'



# Assign a String to a Variable:

- Assigning a string to a variable is done with the variable followed by the <- operator and the string:

**Example:** `str <- "Hello"`

`str # print the value of str`

## Multiline Strings:

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

**Example:**

- `str <- "Hi, Welcome to future connect training institute, here  
you can learn your desire courses with hands on practice."  
str # print the value of str`

- However, note that R will add a **"\n"** at the end of each line break. This is called an **escape character**, and the **n character indicates a new line**.
- If you want the line breaks to be inserted at the same position as in the code, use the **cat()** function:

### Example:

```
str <- "Thank you for contacting us  
our support team will be sort your queries  
as soon as possible ."  
cat(str)
```



# String Length:

- There are many useful string functions in R.
- For example, to find the number of characters in a string, use the **nchar()** function:

## Example:

```
str <- "Hello World!"  
nchar(str)
```

# Check a String:

- Use the **grepl()** function to check if a character or a sequence of characters are present in a string:

## Example:

```
str <- "Hello World!"  
grepl("H", str)  
grepl("Hello", str)  
grepl("X", str)
```

## Combine Two Strings:

- Use the **paste()** function to merge/concatenate two strings:

## Example

```
str1 <- "Hello"  
str2 <- "World"  
paste(str1, str2)
```



# Escape Characters:

- To insert characters that are illegal in a string, you must use an escape character.
- An escape character is a backslash \ followed by the character you want to insert.
- An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

## Example:

```
str <- "We are the so-called "Vikings", from the north."  
str
```

Result:

Error: unexpected symbol in "str <- "We are the so-called "Vikings"

To fix this problem, use the escape character \":

- Note that auto-printing the str variable will print the backslash in the output. You can use the **cat()** function to print it without backslash.
- Other **escape characters** in R:

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



# R Booleans/ Logical values



Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

# 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:



## Example:

```
10 > 9    # TRUE because 10 is greater than 9  
10 == 9   # FALSE because 10 is not equal to 9  
10 < 9    # FALSE because 10 is greater than 9
```

You can also compare two variables:

## Example:

```
a <- 10  
b <- 9  
a > b
```

You can also run a condition in an if statement, which you will learn much more about in the if-else chapter.

## Example:

```
a <- 200
b <- 33
if (b > a) {
  print("b is greater than a")
} else {
  print("b is not greater than a")
}
```



# R Operators



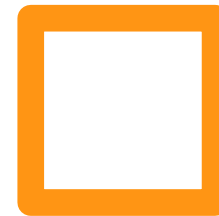
Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

# Operators:

- Operators are used to perform operations on variables and values.
- In the example below, we use the + operator to add together two values:
- Example:

$10 + 5$





# R divides the operators in the following groups:

Arithmetic operators

Assignment operators

Comparison operators

Logical operators

Miscellaneous operators



# R Arithmetic Operators:

- Arithmetic operators are used with numeric values to perform common mathematical operations:



Operator	Name	Example	
+	Addition	$x + y$	
-	Subtraction	$x - y$	
*	Multiplication	$x * y$	
/	Division	$x / y$	
^	Exponent	$x ^ y$	
%%	Modulus (Remainder from division)	$x \% y$	
%/%	Integer Division	$x \% / \% y$	

# R Assignment Operators

- Assignment operators are used to assign values to variables:

## Example:

```
my_var <- 3
my_var <<- 3
3 -> my_var
3 ->> my_var
my_var # print my_var
```

## R Comparison Operators:

- Comparison operators are used to compare two values:

Operator	Name	Example	
==	Equal	x == y	
!=	Not equal	x != y	
>	Greater than	x > y	
<	Less than	x < y	
>=	Greater than or equal to	x >= y	
<=	Less than or equal to	x <= y	

# R Logical Operators:

- Logical operators are used to combine conditional statements:

Operator	Description
&	Element-wise Logical AND operator. It returns TRUE if both elements are TRUE
&&	Logical AND operator - Returns TRUE if both statements are TRUE
	Elementwise- Logical OR operator. It returns TRUE if one of the statement is TRUE
	Logical OR operator. It returns TRUE if one of the statement is TRUE.
!	Logical NOT - returns FALSE if statement is TRUE

# R Miscellaneous Operators:

- Miscellaneous operators are used to manipulate data:

Operator	Description	Example
:	Creates a series of numbers in a sequence	<code>x &lt;- 1:10</code>
<code>%in%</code>	Find out if an element belongs to a vector	<code>x %in% y</code>
<code>%*%</code>	Matrix Multiplication	<code>x &lt;- Matrix1 %*% Matrix2</code>



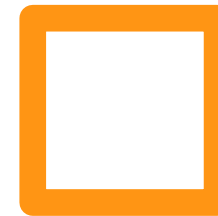
# Conditions and Statements in R



Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

In R, conditions and statements are fundamental elements used to control the flow of a program. Conditions are used to make decisions, and statements are executed based on the outcome of those decisions. Two common constructs for working with conditions and statements in R are:



# Conditions and If Statements:

- R supports the usual logical conditions from mathematics:

Operator	Name	Example	
<code>==</code>	Equal	<code>x == y</code>	
<code>!=</code>	Not equal	<code>x != y</code>	
<code>&gt;</code>	Greater than	<code>x &gt; y</code>	
<code>&lt;</code>	Less than	<code>x &lt; y</code>	
<code>&gt;=</code>	Greater than or equal to	<code>x &gt;= y</code>	
<code>&lt;=</code>	Less than or equal to	<code>x &lt;= y</code>	

These conditions can be used in several ways, most commonly in "if statements" and loops.



# 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**:

## Example:

```
a <- 33
b <- 200
if (b > a) {
  print("b is greater than a")
}
```

- In this example we use two variables, a and b, which are used as a part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".
- R uses curly brackets { } to define the scope in the code.

## Else If:

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

### Example:

```
a <- 33
b <- 33
if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print("a and b are equal")
}
```

- In this example a is equal to b, so the first condition is not true, but the else if condition is true, so we print to screen that "a and b are equal".
- You can use as many else if statements as you want in R.

## If Else:

- The else keyword catches anything which isn't caught by the preceding conditions:

### Example:

```
a <- 200
b <- 33
if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print("a and b are equal")
} else {
  print("a is greater than b")
}
```

- In this example, a is greater than b, so the first condition is not true, also the else if condition is not true, so we go to the else condition and print to screen that "a is greater than b".

- **You can also use else without else if:**

## Example:

```
a <- 200
b <- 33
if (b > a) {
  print("b is greater than a")
} else {
  print("b is not greater than a")
}
```



## Nested If Statements:

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

### Example:

```
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.")
}
```

## AND:

- The & symbol (and) is a logical operator, and is used to combine conditional statements:

### Example:

Test if a is greater than b, AND if c is greater than a:

```
a <- 200
```

```
b <- 33
```

```
c <- 500
```

```
if (a > b & c > a) {  
  print("Both conditions are true")  
}
```

## OR:

The | symbol (or) is a logical operator, and is used to combine conditional statements:

### Example:

Test if a is greater than b, or if c is greater than a:

```
a <- 200
```

```
b <- 33
```

```
c <- 500
```

```
if (a > b | a > c) {  
  print("At least one of the conditions is true")  
}
```



# Loops in R



Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

# Loops:

- 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





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

## Example:

Print i as long as i is less than 6:

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

- In the example above, the loop will continue to produce numbers ranging from 1 to 5. The loop will stop at 6 because  $6 < 6$  is FALSE.
- The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

## Break:

- With the break statement, we can stop the loop even if the while condition is TRUE:

### Example:

Exit the loop if i is equal to 4.

```
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
  if (i == 4) {
    break
  }
}
```

- The loop will stop at 3 because we have chosen to finish the loop by using the break statement when i is equal to 4 (i == 4).

## Next:

---

- With the next statement, we can skip an iteration without terminating the loop:

### Example:

Skip the value of 3:

```
i <- 0
while (i < 6) {
  i <- i + 1
  if (i == 3) {
    next
  }
  print(i)
}
```

- When the loop passes the value 3, it will skip it and continue to loop.

# Yahtzee!

- If .. Else Combined with a While Loop
- To demonstrate a practical example, let us say we play a game of Yahtzee!

## Example:

Print "Yahtzee!" If the dice number is 6:

```
dice <- 1
while (dice <= 6) {
  if (dice < 6) {
    print("No Yahtzee")
  } else {
    print("Yahtzee!")
  }
  dice <- dice + 1
}
```

- If the loop passes the values ranging from 1 to 5, it prints "No Yahtzee". Whenever it passes the value 6, it prints "Yahtzee!".

## For Loops:

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

### Example:

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

- This is less like the for keyword in other programming languages and works more like an iterator method as found in other object-orientated programming languages.

### Nested Loops:

- It is also possible to place a loop inside another loop. This is called a nested loop:

### Example:

```
Print the adjective of each fruit in a list:  
adj <- list("red", "big", "tasty")  
fruits <- list("apple", "banana", "cherry")  
for (x in adj) {  
  for (y in fruits) {  
    print(paste(x, y))  
  }  
}
```



# R Functions



Part of Future Connect Media's IT  
Course

By Abdullah Hashmi

## 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.

### Creating a Function:

- To create a function, use the function() keyword:

### Example:

```
my_function <- function() { # create a function  
  with the name my_function  
  print("Hello World!")  
}
```



## Number of Arguments:

- By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less:

**Example:** This function expects 2 arguments, and gets 2 arguments:

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

- If you try to call the function with 1 or 3 arguments, you will get an error:



## Default Parameter Value:

- The following example shows how to use a default parameter value.
- If we call the function without an argument, it uses the default value:

### Example:

```
my_function <- function(country = "Norway") {  
  paste("I am from", country)  
}  
my_function("Sweden")  
my_function("India")  
my_function() # will get the default value, which is Norway  
my_function("USA")
```

## Return Values:

- To let a function return a result, use the `return()` function:

Example:

```
my_function <- function(x) {  
  return (5 * x)  
}  
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```

## Nested Functions:

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

## Example:

Call a function within another function:

```
Nested_function <- function(x, y) {  
  a <- x + y  
  return(a)  
}  
Nested_function(Nested_function(2,2), Nested_function(3,3))
```

## Recursion:

- R also accepts function recursion, which means a defined function can call itself.
- Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never **terminates**, or one that uses excess amounts of memory or processor power. However, when written correctly, **recursion** can be a very efficient and **mathematically**-elegant approach to programming.

In this example, **tri\_recursion()** is a function that we have defined to call itself ("**recurse**"). We use the **k** variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

## Example:

```
tri_recursion <- function(k) {  
  if (k > 0) {  
    result <- k + tri_recursion(k - 1)  
    print(result)  
  } else {  
    result = 0  
    return(result)  
  }  
}  
tri_recursion(6)
```

# Global Variables:

---

- Variables that are created outside of a function are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside.

## Example:

Create a variable outside of a function and use it inside the function:

```
txt <- "awesome"  
my_function <- function() {  
  paste("R is", txt)  
}  
my_function()
```

- If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

## Example:

Create a variable inside of a function with the same name as the global variable:

```
txt <- "global variable"
my_function <- function() {
  txt = "fantastic"
  paste("R is", txt)
}
my_function()
txt # print txt
```

- If you try to print txt, it will return "global variable" because we are printing txt outside the function.

The background features several decorative elements: a large blue circle on the left containing the text 'Thank you'; a purple circle in the top left; an orange L-shaped line in the top right; an orange L-shaped line in the bottom left; and a blue circle with green curved lines in the bottom center.

Thank you

Future Connect Training Institute

Website: <https://www.fctraining.co.uk/>