

Chapter 1

Basics of R

1. Introduction to R

1.1 What is R

R is a programming language. R is often used for statistical computing and graphical presentation to analyze and visualize data.

Example 1: How to output some text, and how to do a simple calculation in R:

```
"Hello World!"
```

```
5 + 5
```

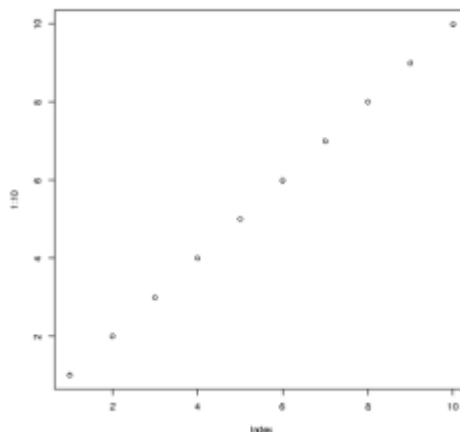
Result:

```
[1] "Hello World!"  
[1] 10
```

Example 2: How you can use R to easily create a graph with numbers from 1 to 10 on both the x and y axis:

```
plot(1:10)
```

Result:



1.2 Why Use R?

- It is a great resource for data analysis, data visualization, data science and machine learning
- It provides many statistical techniques (such as statistical tests, classification, clustering and data reduction)
- It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc++
- It works on different platforms (Windows, Mac, Linux)
- It is open-source and free
- It has a large community support
- It has many packages (libraries of functions) that can be used to solve different problems

2. How to Install R

To install R, go to <https://posit.co/download/rstudio-desktop/> and download the latest version of R and R Studio Windows, Mac or Linux.

- When you have downloaded and installed R, you can run R on your computer.
- If you type `5 + 5`, and press enter, you will see that R outputs `10`



3.R Syntax

3.1 Syntax

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

```
"Hello World!"
```

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

- To output numbers, just type the number (without quotes):

```
5
```

```
10
```

```
25
```

```
[1] 5
[1] 10
[1] 25
```

- To do simple calculations, add numbers together:

```
5 + 5
```

```
[1] 10
```

3.2 R Print Output

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

```
"Hello World!"
```

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

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 uses the `print()` function to output code.

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

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

And there are times you must use the `print()` function to output code, for example when working with `for` loops

```
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
```

4. R Comments

Comments

- Comments can be used to explain R code, and to make it more readable.
- It can also be used to prevent execution when testing alternative code.
- Comments starts with a `#`. When executing code, R will ignore anything that starts with `#`.

This example uses a comment before a line of code:

Example 1

```
# This is a comment  
"Hello World!"
```

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

Example 2: This example uses a comment at the end of a line of code

```
"Hello World!" # This is a comment
```

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

Multiline Comments

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:

```
# This is a comment  
# written in  
# more than just one line  
"Hello World!"
```

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

5. R Variables

5.1 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:

Example 1

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

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

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

```
name <- "John Doe"

name # auto-print the value of the name variable
```

```
[1] "John Doe"
```

Example 2

```
name <- "John Doe"

print(name) # print the value of the name variable
```

5.2 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 1

```
text <- "awesome"

paste("R is", text)
```

```
[1] "R is awesome"
```

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

```
text1 <- "R is"  
text2 <- "awesome"
```

```
paste(text1, text2)
```

```
[1] "R is awesome"
```

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

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

```
num1 + num2
```

```
[1] 15
```

If you try to combine a string (text) and a number, R will give you an error:

```
num <- 5  
text <- "Some text"
```

```
num + text
```

Result:

```
Error in num + text : non-numeric argument to binary operator
```

5.3 Multiple Variables

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

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

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

```
[1] "Orange"
[1] "Orange"
[1] "Orange"
```

5.4 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!

6. R Data Types

In 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:

```
my_var1 <- 30 # my_var is type of numeric
my_var2 <- "Sally" # my_var is now of type character (aka string)

my_var1 # print my_var1
my_var2 # print my_var2
```

```
[1] 30
[1] "Sally"
```

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:

```
# 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)
```

```
[1] "numeric"
[1] "integer"
[1] "complex"
[1] "character"
[1] "logical"
```

7 R Numbers

There are three number types in R:

- **numeric**

- integer
- complex

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

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

7.1 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"
```

7.2 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"
```

7.3 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"
```

7.4 Type Conversion

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

- `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
```

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

```
[1] 1
[1] 2
[1] "numeric"
[1] "integer"
```

8. R Math

Simple Math: In R, you can use **operators** to perform common mathematical operations on numbers.

The **+** operator is used to add together two values:

```
10 + 5
```

```
[1] 15
```

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

```
10 - 5
```

```
[1] 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:

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

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

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

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

```
sqrt(16)
```

```
[1] 4
```

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

```
abs(-4.7)
```

```
[1] 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:

```
ceiling(1.4)
```

```
floor(1.4)
```

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

```
ceiling(1.1)
```

```
floor(0.9)
```

```
[1] 2
[1] 0
```

9. R Strings

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':

```
"hello"
```

```
'hello'
```

```
[1] "hello"
[1] "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:

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

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

Multiline Strings: 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 # print the value of str
```

```
[1] "Lorem ipsum dolor sit amet,\nconsectetur adipiscing elit,\nsed do eiusmod tempor incididunt\nut labore et dolore magna aliqua."
```

```
str <- "Department of,  
Computer Science and Engineering-  
Data Sciene,  
ATME College of Engineering,  
Mysuru."
```

```
str
```

```
[1] "Department of,\nComputer Science and Engineering-,\nData Sciene,\nATME College of Engineering,\nMysuru"
```

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:

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

```
cat(str)
```

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

```
str <- "Dr Anitha D B,Associate Professor,  
Department of CSE-Data Science,  
ATME College of Engineering, Mysuru."  
  
cat(str)
```

```
Dr Anitha D B,Associate Professor,  
Department of CSE-Data Science,  
ATME College of Engineering, Mysuru.
```

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:

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

```
[1] 12
```

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

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

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

Combine Two Strings: Use the `paste()` function to merge/concatenate two strings:

```
str1 <- "Hello"  
str2 <- "World"
```

```
paste(str1, str2)
```

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

R 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:

```
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 `\"`:

The escape character allows you to use double quotes when you normally would not be allowed:

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

```
str
```

```
cat(str)
```

```
[1] "We are the so-called \"Vikings\", from the north."  
We are the so-called "Vikings", from the north.
```

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

10 R 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 # 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
```

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

You can also compare two variables:

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

```
a > b
```

```
[1] TRUE
```

You can also run a condition in an **if** statement, which you will learn much more about in the [if..else](#) chapter.

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

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



```
[1] "b is not greater than a"
```

11. R Operators

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

```
10 + 5
```

```
[1] 15
```

R divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Miscellaneous operators

11.1 R Arithmetic Operators

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

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

11.2 R Assignment Operators

Assignment operators are used to assign values to variables:

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

```
[1] 3
```

Note: `<<-` is a global assigner.

It is also possible to turn the direction of the assignment operator.

`x <- 3` is equal to `3 -> x`

11.3 R Comparison Operators

Comparison operators are used to compare two values:

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

11.4 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

11.5 R Miscellaneous Operators

Miscellaneous operators are used to manipulate data:

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

12 R If ... Else

Conditions and If Statements

R supports the usual logical conditions from mathematics:

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

```
5 == 5 # TRUE because 5 is equal to 5
5 == 3 # FALSE because 5 is not equal to 3
```

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

```
5 != 3 # returns TRUE because 5 is not equal to 3
```

```
[1] TRUE
```

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

11.1 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**:

```
a <- 33
b <- 200

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

```
[1] "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"

```
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"
```

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:

```
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")
}
```

```
[1] "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**:

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

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

```

} else {
  print("b is not greater than a")
}

```

```
[1] "b is not greater than a"
```

11.2 R Nested If

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!"
```

11.3 R - AND OR Operators

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")
}

```

```
[1] "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")
}
```

```
[1] "At least one of the conditions is true"
```

12.R - 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

12.1 R While 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
}
```

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

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.

Note: remember to increment **i**, or else the loop will continue forever.

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
  }
}
```

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

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)
}
```

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


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

12.2 R 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
```

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.

With the **for** loop we can execute a set of statements, once for each item in a vector, array, list, etc..

Example

Print every item in a list:

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

```
for (x in fruits) {  
  print(x)  
}
```

```
[1] "apple"  
[1] "banana"  
[1] "cherry"
```

Example

Print the number of dices:

```
dice <- c(1, 2, 3, 4, 5, 6)
```

```
for (x in dice) {  
  print(x)  
}
```

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

Break: With the **break** statement, we can stop the loop before it has looped through all the items

Example

Stop the loop at "cherry":

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

```
for (x in fruits) {  
  if (x == "cherry") {  
    break  
  }  
  print(x)  
}
```

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

The loop will stop at "cherry" because we have chosen to finish the loop by using the **break** statement when **x** is equal to "cherry" (**x == "cherry"**).

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

Example

Skip "banana":

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

```
for (x in fruits) {  
  if (x == "banana") {  
    next  
  }  
  print(x)  
}
```

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

When the loop passes "banana", it will skip it and continue to loop.

R 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))
  }
}
```

```
[1] "red apple"
[1] "red banana"
[1] "red cherry"
[1] "big apple"
[1] "big banana"
[1] "big cherry"
[1] "tasty apple"
[1] "tasty banana"
[1] "tasty cherry"
```

13. 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

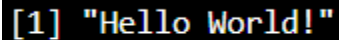
```
my_function <- function() { # create a function with the name my_function
  print("Hello World!")
}
```

Call a Function: To call a function, use the function name followed by parenthesis, like `my_function()`

Example

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

`my_function()` # call the function named my_function

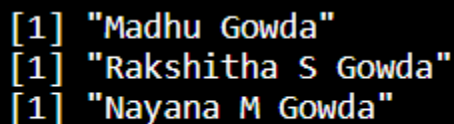


```
[1] "Hello World!"
```

Arguments: Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma. The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name.

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

```
my_function("Madhu")  
my_function("Rakshitha S")  
my_function("Nayana M")
```



```
[1] "Madhu Gowda"  
[1] "Rakshitha S Gowda"  
[1] "Nayana M Gowda"
```

Parameters or Arguments?: The terms "parameter" and "argument" can be used for the same thing: information that are passed into a function.

From a function's perspective: A parameter is the variable listed inside the parentheses in the function definition. An argument is the value that is sent to the function when it is called.

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("Madhu", "Gowda")
```

```
[1] "Madhu Gowda"
```

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

Example

This function expects 2 arguments, and gets 1 argument:

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

```
my_function("Madhu")
```

```
Error in paste(fname, lname) :  
  argument "lname" is missing, with no default  
Calls: my_function -> print -> paste  
Execution halted
```

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")
```

```
[1] "I am from Sweden"  
[1] "I am from India"  
[1] "I am from Norway"  
[1] "I am from 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))
```

The output of the code above will be:

```
[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.

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))
```

```
[1] 10
```

Explanation of the Example

The function tells x to add y.

The first input Nested_function(2,2) is "x" of the main function.

The second input Nested_function(3,3) is "y" of the main function.

The output is therefore $(2+2) + (3+3) = 10$.

Example

Write a function within a function:

```
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
```

Example Explained

You cannot directly call the function because the Inner_func has been defined (nested) inside the Outer_func.

We need to call Outer_func first in order to call Inner_func as a second step.

We need to create a new variable called output and give it a value, which is 3 here.

We then print the output with the desired value of "y", which in this case is 5.

The output is therefore **8** (3 + 5).

R Function 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.

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.

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

```
[1] 1
[1] 3
[1] 6
[1] 10
[1] 15
[1] 21
```

R 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()
```

```
[1] "R is awesome"
```

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
```

```
[1] "R is fantastic"
[1] "global variable"
```

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

The Global Assignment Operator: Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function. To create a global variable inside a function, you can use the **global assignment** operator `<<-`

Example

If you use the assignment operator `<<-`, the variable belongs to the global scope:

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

```
my_function()
```

```
print(txt)
```

```
[1] "R is fantastic"
[1] "fantastic"
```

Also, use the **global** assignment operator if you want to change a global variable inside a function:

Example

To change the value of a global variable inside a function, refer to the variable by using the global assignment operator `<<-`:

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

```
my_function()
```

```
paste("R is", txt)
```

```
[1] "R is fantastic"
[1] "R is fantastic"
```

14. R Vectors

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:

Example

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

```
# Print fruits
fruits
```

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

In this example, we create a vector that combines numerical values:

Example

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

```
# Print numbers
numbers
```

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

To create a vector with numerical values in a sequence, use the `:` operator:

Example

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

```
numbers
```

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

You can also create numerical values with decimals in a sequence, but note that if the last element does not belong to the sequence, it is not used:

Example

```
# Vector with numerical decimals in a sequence
numbers1 <- 1.5:6.5
numbers1
```

Vector with numerical decimals in a sequence where the last element is not used

```
numbers2 <- 1.5:6.3
```

```
numbers2
```

Result:

```
[1] 1.5 2.5 3.5 4.5 5.5 6.5
[1] 1.5 2.5 3.5 4.5 5.5
```

In the example below, we create a vector of logical values:

Example

Vector of logical values

```
log_values <- c(TRUE, FALSE, TRUE, FALSE)
```

```
log_values
```

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

Vector Length: To find out how many items a vector has, use the `length()` function:

Example

```
fruits <- c("banana", "apple", "orange")
```

```
length(fruits)
```

```
[1] 3
```

Sort a Vector: To sort items in a vector alphabetically or numerically, use the `sort()` function:

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
```

```
numbers <- c(13, 3, 5, 7, 20, 2)
```

```
sort(fruits) # Sort a string
```

```
sort(numbers) # Sort numbers
```

```
[1] "apple" "banana" "lemon" "mango" "orange"
[1] 2 3 5 7 13 20
```

Access Vectors: You can access the vector items by referring to its index number inside brackets []. The first item has index 1, the second item has index 2, and so on:

Example

```
fruits <- c("banana", "apple", "orange")
```

```
# Access the first item (banana)
fruits[1]
```

```
[1] "banana"
```

You can also access multiple elements by referring to different index positions with the c() function:

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
```

```
# Access the first and third item (banana and orange)
fruits[c(1, 3)]
```

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

You can also use negative index numbers to access all items except the ones specified:

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
```

```
# Access all items except for the first item
fruits[c(-1)]
```

```
[1] "apple" "orange" "mango" "lemon"
```

Change an Item: To change the value of a specific item, refer to the index number

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")  
  
# Change "banana" to "pear"  
fruits[1] <- "pear"  
  
# Print fruits  
fruits
```

```
[1] "pear" "apple" "orange" "mango" "lemon"
```

Repeat Vectors: To repeat vectors, use the `rep()` function:

Example

Repeat each value:

```
repeat_each <- rep(c(1,2,3), each = 3)  
  
repeat_each
```

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

Example

Repeat the sequence of the vector:

```
repeat_times <- rep(c(1,2,3), times = 3)  
  
repeat_times
```

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

Example

Repeat each value independently:

```
repeat_indepent <- rep(c(1,2,3), times = c(5,2,1))  
  
repeat_indepent
```

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

Generating Sequenced Vectors: One of the examples on top, showed you how to create a vector with numerical values in a sequence with the `:` operator:

Example

```
numbers <- 1:10
```

```
numbers
```

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

To make bigger or smaller steps in a sequence, use the `seq()` function:

Example

```
numbers <- seq(from = 0, to = 100, by = 20)
```

```
numbers
```

```
[1] 0 20 40 60 80 100
```

Note: The `seq()` function has three parameters: `from` is where the sequence starts, `to` is where the sequence stops, and `by` is the interval of the sequence.