Numeric data are numbers that contain a decimal

L integer 125L

125 numeric

FALSE, TRUE, F, T Logical Literals

character literals include:

* "Hello"

3 x 4 wrong, 3\*4

* Key R syntax principles: R is case-sensitive and permits only one instruction per line of code.
* Modulo (%%): This operator returns the remainder after an integer division. Continuing the above example, the remainder when 17 is divided by 5 is 2. Hence, 17 %% 5 produces 2.
* Integer Division (%/%): The operation x %/% n determines how many times the number n can be found within x without exceeding it. For example, 17L %/% 5L results in 3, indicating that 5 can be found three times in 17 without surpassing it.
* Modulo (%%): This operator returns the remainder after an integer division. Continuing the above example, the remainder when 17 is divided by 5 is 2. Hence, 17 %% 5 produces 2.

x <- 42

txt <- "World!"

("Hello", )

for (x in 1:10) {

print(x)

}

name <- "John"  
age <- 40

name <- "John Doe"  
  
print(name) # print the value of the name variable

for (x in 1:10) {  
  print(x)  
}

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

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

num1 <- 5  
num2 <- 10  
  
num1 + num2

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

# 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"

my\_var <- 30 # my\_var is type of **numeric**  
my\_var <- "Sally" # my\_var is now of type **character** (aka string)

* 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)

# 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)

x <- 1000L  
y <- 55L  
  
# Print values of x and y  
x  
y  
  
# Print the class name of x and y  
class(x)  
class(y)

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)

max(5, 10, 15)  
  
min(5, 10, 15)

sqrt(16)

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) -----------2  
  
floor(1.4)---------------1

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

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

If you want the line breaks to be inserted at the same position as in the code, use the cat() function:

Example

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

to find the number of characters in a string, use the nchar() function:

Example

str <- "Hello World!"  
  
nchar(str)

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)

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.

str <- "We are the so-called \"Vikings\", from the north."  
  
str  
cat(str)

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

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

a <- 10  
b <- 9  
  
a > b

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

<<- is a global assigner.

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 | x <- 1:10 |
| %in% | Find out if an element belongs to a vector | x %in% y |
| %\*% | Matrix Multiplication | x <- Matrix1 %\*% Matrix2 |

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

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.")  
}

a <- 200  
b <- 33  
c <- 500  
  
if (a > b & c > a) {  
  print("Both conditions are true")  
}

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  
}

---------------------------------------------------------

Exit the loop if i is equal to 4.

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

------------------------------------------------------

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

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

for (x in 1:10) {  
  print(x)  
}

fruits <- list("apple", "banana", "cherry")  
  
for (x in fruits) {  
  print(x)  
}

dice <- c(1, 2, 3, 4, 5, 6)  
  
for (x in dice) {  
  print(x)  
}

fruits <- list("apple", "banana", "cherry")  
  
for (x in fruits) {  
  if (x == "cherry") {  
    break  
  }  
  print(x)  
}

fruits <- list("apple", "banana", "cherry")  
  
for (x in fruits) {  
  if (x == "banana") {  
    next  
  }  
  print(x)  
}

dice <- 1:6  
  
for(x in dice) {  
  if (x == 6) {  
    print(paste("The dice number is", x, "Yahtzee!"))  
  } else {  
    print(paste("The dice number is", x, "Not Yahtzee"))  
  }  
}

adj <- list("red", "big", "tasty")  
  
fruits <- list("apple", "banana", "cherry")  
  for (x in adj) {  
    for (y in fruits) {  
      print(paste(x, y))  
  }  
}

<https://www.w3schools.com/r/r_functions.asp>

my\_function <- function() { # create a function with the name my\_function  
  print("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:

### Example

my\_function <- function(fname) {  
  paste(fname, "Griffin")  
}  
  
my\_function("Peter")  
my\_function("Lois")  
my\_function("Stewie")

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.

This function expects 2 arguments, and gets 2 arguments:

my\_function <- function(fname, lname) {  
  paste(fname, lname)  
}  
  
my\_function("Peter", "Griffin")

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")

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))

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))

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**.

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)

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).

## **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)

https://www.w3schools.com/r/r\_functions\_recursion.asp