# R Variables and Constants

you will learn about R variables and constants with the help of examples.

In computer programming, a variable is a named memory location where data is stored. For example,

```
x = 13.8
```

Here, `x` is the variable where the data **13.8** is stored. Now, whenever we use `x` in our program, we will get **13.8**.

```
x = 13.8

# print variable
print(x)
```

**Output**

```
[1] 13.8
```

As you can see, when we print `x` we get `13.8` as output.

---

## Rules to Declare R Variables

As per our requirements, we can use any name for our variables. However, there are certain rules that need to be followed while creating a variable:

- A variable name in R can be created using letters, digits, periods, and underscores.
- You can start a variable name with a letter or a period, but not with digits.
- If a variable name starts with a dot, you can't follow it with digits.

- R is case sensitive. This means that `age` and `Age` are treated as different variables.
- We have some reserved words that cannot be used as variable names.

> **Note**: In earlier versions of R programming, the period `.` was used to join words in a multi-word variable such as `first.name`, `my.age`, etc. However, nowadays we mostly use `_` for multi-word variables For example, `first_name`, `my_age`, etc.

---

## Types of R Variables

Depending on the type of data that you want to store, variables can be divided into the following types.

### 1. Boolean Variables

It stores single bit data which is either `TRUE` or `FALSE`. Here, `TRUE` means yes and `FALSE` means no. For example,

```
a = TRUE

print(a)
print(class(a))
```

**Output**

```
[1] TRUE
[1] "logical"
```

Here, we have declared the boolean variable `a` with the value `TRUE`. Boolean variables belong to the logical class so `class(a)` returns `"logical"`.

---

## 2. Integer Variables

It stores numeric data without any decimal values. For example,

```
A = 14L

print(A)
print(class(A))
```

**Output**

```
[1] 14
[1] "integer"
```

Here, `L` represents integer value. In R, integer variables belong to the integer class so, `class(a)` returns `"integer"`.

---

## 3. Floating Point Variables

It stores numeric data with decimal values. For example,

```
x = 13.4

print(x)
print(class(x))
```

**Output**

```
[1] 13.4
[1] "numeric"
```

Here, we have created a floating point variable named `x`. You can see that the floating point variable belongs to the `numeric` class.

---

## 4. Character Variables

It stores a single character data. For example,

```
alphabet = "a"

print(alphabet)
print(class(alphabet))
```

**Output**

```
[1] "a"
[1] "character"
```

Here, we have created a character variable named `alphabet`. Since character variables belong to the character class, `class(alphabet)` returns `"character"`.

---

## 5. String Variables

It stores data that is composed of more than one character. We use double quotes to represent string data. For example,

```
message = "Welcome to Programiz!"

print(message)
print(class(message))
```

**Output**

```
[1] "Welcome to Programiz!"
[1] "character"
```

Here, we have created a string variable named `message`. You can see that the string variable also belongs to the `character` class.

---

## Changing Value of Variables

Depending on the conditions or information passed into the program, you can change the value of a variable. For example,

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

# changing value of a variable
message <- "Welcome to Programiz!"


print(message)
```

**Output**

```
[1] "Hello World!"
[1] "Welcome to Programiz!"
```

In this program,

- `"Hello World!"` - initial value of `message`
- `"Welcome to Programiz!"` - changed value of `message`

You can see that the value of a variable can be changed anytime.

## Built-In R Constants

R programming provides some predefined constants that can be directly used in our program. For example,

```
# print list of uppercase letters
print(LETTERS)

# print list of lowercase letters
print(letters)

# print 3 letters abbreviation of English months
print(month.abb)

# print numerical value of constant pi
print(pi)
```

**Output**

```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R"
"S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
"s"
[20] "t" "u" "v" "w" "x" "y" "z"
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
[1] 3.141593
```

# R Print Output

In this tutorial, you will learn how to use different methods to print output in the console in R.

## R print() Function

In R, we use the `print()` function to print values and variables. For example,

```
# print values
print("R is fun")


# print variables
x <- "Welcome to Programiz"
print(x)
```

**Output**

```
[1] "R is fun"
[1] "Welcome to Programiz"
```

In the above example, we have used the `print()` function to print a string and a variable. When we use a variable inside `print()`, it prints the value stored inside the variable.

---

## paste() Function in R

You can also print a string and variable together using the `print()` function. For this, you have to use the `paste()` function inside `print()`. For example,

```
company <- "Programiz"

# print string and variable together
print(paste("Welcome to", company))
```

**Output**

```
Welcome to Programiz
```

Notice the use of the `paste()` function inside `print()`. The `paste()` function takes two arguments:

- **string** - "Welcome to"
- **variable** - company

By default, you can see there is a space between string `Welcome to` and the value `Programiz`.

If you don't want any default separator between the string and variable, you can use another variant of `paste()` called `paste0()`. For example,

```
company <- "Programiz"

# using paste0() instead of paste()
print(paste0("Welcome to", company))
```

**Output**

```
[1] "Welcome toProgramiz"
```

Now, you can see there is no space between the string and the variable.

---

## R sprintf() Function

The `sprintf()` function of C Programming can also be used in R. It is used to print formatted strings. For example,

```
myString <- "Welcome to Programiz"

# print formatted string
sprintf("String: %s", myString)
```

**Output**

```
[1] "String: Welcome to Programiz"
```

Here,

- `String: %s` - a formatted string
- `%s` - format specifier that represents string values
- `myString` - variable that replaces the format specifier `%s`

Besides `%s`, there are many other format specifiers that can be used for different types values.

| Specifier | Value Type |
|---|---|
| `%c` | Character |
| `%d` or `%i` | Signed Decimal Integer |
| `%e` or `%E` | Scientific notation |
| `%f` | Decimal Floating Point |
| `%u` | Unsigned Decimal Integer |
| `%p` | Pointer address |

Let's use some of them in examples.

```
# sprintf() with integer variable
myInteger <- 123
sprintf("Integer Value: %d", myInteger)


# sprintf() with float variable
myFloat <- 12.34
```

```
sprintf("Float Value: %f", myFloat)
```

### Output

```
[1] "Integer Value: 123"
[1] "Float Value: 12.340000"
```

---

## R cat() Function

R programming also provides the `cat()` function to print variables. However, unlike `print()`, the `cat()` function is only used with basic types like logical, integer, character, etc.

```
# print using Cat
cat("R Tutorials\n")


# print a variable using Cat
message <- "Programiz"
cat("Welcome to ", message)
```

### Output

```
R Tutorials
Welcome to  Programiz
```

In the example above, we have used the `cat()` function to display a string along with a variable. The `\n` is used as a newline character.

> **Note**: As mentioned earlier, you cannot use `cat()` with list or any other object.

---

## Print Variables in R Terminal

You can also print variables inside the R terminal by simply typing the variable name. For example,

```
# inside R terminal
x = "Welcome to Programiz!"

# print value of x in console
x

// Output: [1] "Welcome to Programiz"
```

# R Booleans (Comparison and Logical Operators)

In this tutorial, you will learn in detail about R booleans with the help of comparison and logical operators.

In R, boolean variables can take only 2 values: `TRUE` and `FALSE`. For example,

```
# declare boolean
x <- TRUE

print(x)
print(class(x))

# declare boolean using single character
y <- F

print(y)
print(class(y))
```

### Output

```
[1] TRUE
[1] "logical"
```

```
[1] FALSE
[1] "logical"
```

Here, we have declared `x` and `y` as boolean variables. In R, Boolean variables belong to the `logical` class.

You can also declare boolean variables using a single character - `T` or `F`. Here, `T` stands for `TRUE` and `F` stands for `FALSE`.

---

## R Boolean With Comparison Operators

Comparison operators are used to compare two values.

| Operator | Description | Example |
|---|---|---|
| `>` | Greater than | `5 > 6` returns `FALSE` |
| `<` | Less than | `5 < 6` returns `TRUE` |
| `==` | Equals to | `10 == 10` returns `TRUE` |
| `!=` | Not equal to | `10 != 10` returns `FALSE` |
| `>=` | Greater than or equal to | `5 >= 6` returns `FALSE` |
| `<=` | Less than or equal to | `6 <= 6` returns `TRUE` |

The output of a comparison is a boolean value. For example, to check if two numbers are equal, you can use the `==` operator.

```
x <- 10
y <- 23

# compare x and y
print(x == y)  # FALSE
```

Similarly, to check if `x` is less than `y`, you can use the `<` operator.

```
x <- 10
```

```
y <- 23

# compare x and y
print(x < y)  # TRUE
```

Since, the value stored in `x` is less than the value stored in `y`, the comparison `x < y` results in `TRUE`.

---

## R Boolean With Logical Operators

Logical operators are used to compare the output of two comparisons. There are three types of logical operators in R. They are:

- AND operator (`&`)
- OR operator (`|`)
- NOT operator (`!`)

---

### AND Operator (&)

The AND operator `&` takes as input two logical values and returns the output as another logical value.
The output of the operator is `TRUE` only when both the input logical values are either `TRUE` or evaluated to `TRUE`.
Let `a` and `b` represent two operands. `0` represents `FALSE` and `1` represents `TRUE`. Then,

| a | b | a & b |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

For example,

```
# print & of TRUE and FALSE combinations
TRUE & TRUE
TRUE & FALSE
FALSE & TRUE
FALSE & FALSE
```

**Output**

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

The input to any logical operator can also be a comparison between two or more variables. For example,

```
x <- 10
y <- 23
z <- 12

print(x<y & y>z)
```

**Output**

```
[1] TRUE
```

Here, the condition checks whether `x` is less than `y` and `y` is less than `z` or not. If both the conditions evaluate to `TRUE`, then the output is `TRUE`. If any of them is `FALSE`, the output turns out to be `FALSE`.

---

## OR Operator (|)

The OR operator `|` returns `TRUE` if all or any one of the logical inputs is `TRUE` or evaluates to `TRUE`. If all of them are `FALSE`, then it returns `FALSE`.

Consider the table below.

| a | b | a \| b |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

For example,

```
# print | of TRUE and FALSE combinations
TRUE | TRUE
TRUE | FALSE
FALSE | TRUE
FALSE | FALSE
```

**Output**

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

Here, if any one of the inputs is `TRUE`, then the output is `TRUE`.

Similar to the case of AND operator, you can use any number of comparisons as input to the OR operator. For example,

```
w <- 54
x <- 12
y <- 25
z <- 1

print(w>x | x>y | z>w)
```

**Output**

```
[1] TRUE
```

Here, only the comparison `w>x` evaluates to `TRUE`. Apart from that, all the other comparisons evaluate to `FALSE`. Since, at least one of the inputs is `TRUE`, the output of the entire comparison is `TRUE`.

---

## NOT (!) Operator

The NOT operator `!` is used to negate the logical values it is used on. If the input value is `TRUE`, it will turn to `FALSE` and vice-versa.

| a | !a |
|---|---|
| 1 | 0 |
| 0 | 1 |

For example,

```
# print ! of TRUE and FALSE
!TRUE
!FALSE
```

**Output**

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

Here, the output is the negation of the input.

We can use the `!` operator with comparisons. For example, `!(x > 12)` is the same as `x <= 12`. This means that `x` is not greater than **12**. Which means that `x` can be less than or equal to **12**.

---

You can also use the NOT operator with any in-built function that evaluates to boolean value. For example,

```
x <- 3 + 5i

# using ! with in-built function
print(!is.numeric(x))
```

**Output**

```
[1] TRUE
```

Here, since `x` is of type `complex`, the function `is.numeric(x)` evaluates to `FALSE` and the negation of `FALSE` is `TRUE`, hence the output.

---

## Example: R Comparison and Logical Operators

You can use all the three logical operators with comparison operators.

```
x <- 5

print(is.numeric(x) & (x>5 | x==5))
```

**Output**

```
[1] TRUE
```

Here, we can consider the entire operation in two parts - `is.numeric(x)` and `(x>5 | x==5)`. Since, there is an AND operator between them, if both of them evaluate to `TRUE`, only then the output is `TRUE`.
This is how the program works:

- `is.numeric(x)` - this evaluates to `TRUE` since `x` is of `numeric` type
- `(x>5 | x==5)` - this evaluates to `TRUE` since `x==5` is `TRUE`

# R if...else

In this tutorial, you will learn about if...else statements in R with the help of examples.

In computer programming, the if statement allows us to create a decision making program.

A decision making program runs one block of code under a condition and another block of code under different conditions. For example,

- If **age is greater than 18**, allow the person to vote.
- If **age is not greater than 18**, don't allow the person to vote.

---

# R if Statement

The syntax of an if statement is:

```
if (test_expression) {
  # body of if
}
```

Here, the `test_expression` is a boolean expression. It returns either `True` or `False`. If the `test_expression` is

- **True** - body of the if statement is executed
- **False** - body of the if statement is skipped

---

### Example: R if Statement

```
x <- 3

# check if x is greater than 0
if (x > 0) {
  print("The number is positive")
}

print("Outside if statement")
```

**Output**

```
[1] "The number is positive"
[1] "Outside if statement"
```

In the above program, the test condition x > 0 is true. Hence, the code inside parenthesis is executed.

> **Note:** If you want to learn more about test conditions, visit R Booleans Expression.

---

## R if...else Statement

We can also use an optional else statement with an if statement. The syntax of an if...else statement is:

```
if (test_expression) {
  # body of if statement
} else {
  # body of else statement
}
```

The if statement evaluates the test_expression inside the parentheses.

**If the test_expression is True,**

- body of `if` is executed

- body of `else` is skipped

**If the test_expression is False**

- body of `else` is executed
- body of `if` is skipped

---

## Example: R if...else Statement

```
age <- 15

# check if age is greater than 18
if (age > 18) {
  print("You are eligible to vote.")
} else {
  print("You cannot vote.")
}
```

**Output**

```
[1] "You cannot vote."
```

In the above statement, we have created a variable named `age`. Here, the test expression is

```
age > 18
```

Since `age` is **16**, the test expression is `False`. Hence, code inside the `else` statement is executed.

If we change the variable to another number. Let's say **31**.

```
age <- 31
```

Now, if we run the program, the output will be:

```
[1] "You are eligible to vote."
```

---

### Example: Check Negative and Positive Number

```
x <- 12

# check if x is positive or negative number
if (x > 0) {
  print("x is a positive number")
} else {
  print("x is a negative number")
}
```

### Output

```
[1] "x is a positive number"
```

Here, since `x > 0` evaluates to `TRUE`, the code inside the `if` block gets executed. And, the code inside the else block is skipped.

---

## R if...else if...else Statement

If you want to test more than one condition, you can use the optional `else if` statement along with your `if...else` statements. The syntax is:

```
if(test_expression1) {
  # code block 1
} else if (test_expression2){
  # code block 2
} else {
  # code block 3
}
```

Here,

- If **test_expression1** evaluates to `True`, the **code block 1** is executed.
- If **test_expression1** evaluates to `False`, then **test_expression2** is evaluated.
  - If **test_expression2** is `True`, **code block 2** is executed.
  - If **test_expression2** is `False`, **code block 3** is executed.

### Example: R if...else if...else Statement

```
x <- 0

# check if x is positive or negative or zero
if (x > 0) {
  print("x is a positive number")
} else if (x < 0) {
  print("x is a negative number")
} else {
  print("x is zero")
}
```

### Output

```
[1] "x is zero"
```

In the above example, we have created a variable named `x` with the value **0**. Here ,we have two test expressions:

- `if (x > 0)` - checks if `x` is greater than **0**
- `else if (x < 0)` - checks if `x` is less than **0**

Here, both the test conditions are `False`. Hence, the statement inside the body of `else` is executed.

---

## Nested if...else Statements in R

You can have nested `if...else` statements inside `if...else` blocks in R. This is called nested if...else statement.

This allows you to specify conditions inside conditions. For example,

```
x <- 20

# check if x is positive
if (x > 0) {
```

```
  # check if x is even or odd
  if (x %% 2 == 0) {
    print("x is a positive even number")
  } else {
    print("x is a positive odd number")
  }

# execute if x is not positive
} else {

  # check if x is even or odd
  if (x %% 2 == 0) {
    print("x is a negative even number")
  } else {
    print("x is a negative odd number")
  }
}
```

**Output**

```
[1] "x is a positive even number"
```

In this program, the outer `if...else` block checks whether `x` is positive or negative. If `x` is greater than **0**, the code inside the outer `if` block is executed.

Otherwise, the code inside the outer `else` block is executed.

```
if (x > 0) {
  ... .. ...
} else {
  ... .. ...
}
```

The inner `if...else` block checks whether `x` is even or odd. If `x` is perfectly divisible by **2**, the code inside the inner `if` block is executed. Otherwise, the code inside the inner `else` block is executed.

```
if (x %% 2 == 0) {
  ... .. ...
} else {
  ... .. ...
}
```

# R ifelse() Function

In this tutorial, you will learn about the ifelse() function in R with examples.

In R, the `ifelse()` function is a shorthand vectorized alternative to the standard `if...else` statement.
Most of the functions in R take a vector as input and return a vectorized output. Similarly, the vector equivalent of the traditional `if...else` block is the `ifelse()` function.
The syntax of the `ifelse()` function is:

```
ifelse(test_expression, x, y)
```

The output vector has the element `x` if the output of the `test_expression` is `TRUE`. If the output is `FALSE`, then the element in the output vector will be `y`.

---

## Example 1: ifelse() Function for Odd/Even Numbers

```
# input vector
x <- c(12, 9, 23, 14, 20, 1, 5)

# ifelse() function to determine odd/even numbers
ifelse(x %% 2 == 0, "EVEN", "ODD")
```

**Output**

```
[1] "EVEN" "ODD"  "ODD"  "EVEN" "EVEN" "ODD"  "ODD"
```

In this program, we have defined a vector `x` using the `c()` function in R. The vector contains a few odd and even numbers.

We then used the `ifelse()` function which takes the vector `x` as an input. A logical operation is then performed on `x` to determine if the elements are odd or even.

For each element in the vector, if the `test_expression` evaluates to `TRUE`, then the corresponding output element is "`EVEN`", else it's "`ODD`".

---

## Example 2: ifelse() Function for Pass/Fail

```
# input vector of marks
marks <- c(63, 58, 12, 99, 49, 39, 41, 2)

# ifelse() function to determine pass/fail
ifelse(marks < 40, "FAIL", "PASS")
```

**Output**

```
[1] "PASS" "PASS" "FAIL" "PASS" "PASS" "FAIL" "PASS" "FAIL"
```

This program determines if the students have passed or failed based on a condition. Here, if the marks in the vector are less than **40**, then the student is considered to have failed

# R while Loop

In this tutorial, you will learn about while loops in R with the help of examples.

In programming, loops are used to repeat a block of code as long as the specified condition is satisfied. Loops help you to save time, avoid repeatable blocks of code, and write cleaner code.

In R, there are three types of loops:

- while loops

- for loops
- *repeat loops*

---

## R while Loop

`while` loops are used when you don't know the exact number of times a block of code is to be repeated. The basic syntax of `while` loop in R is:

```
while (test_expression) {
  # block of code
}
```

- Here, the `test_expression` is first evaluated.
- If the result is `TRUE`, then the block of code inside the `while` loop gets executed.
- Once the execution is completed, the `test_expression` is evaluated again and the same process is repeated until the `test_expression` evaluates to `FALSE`.
- The `while` loop will terminate when the boolean expression returns `FALSE`.

---

### Example 1: R While Loop

Let's look at a program to calculate the sum of the first ten natural numbers.

```
# variable to store current number
number = 1

# variable to store current sum
sum = 0

# while loop to calculate sum
```

```
while(number <= 10) {

  # calculate sum
  sum = sum + number

  # increment number by 1
  number = number + 1
}


print(sum)
```

**Output**

```
[1] 55
```

Here, we have declared two variables: `number` and `sum`.
The `test_condition` inside the `while` statement is `number <= 10`.
This means that the `while` loop will continue to execute and calculate the `sum` as long as the value of `number` is less than or equal to `10`.

---

## Example 2: while Loop With break Statement

The `break` statement in R can be used to stop the execution of a `while` loop even when the test expression is `TRUE`. For example,

```
number = 1

# while loop to print numbers from 1 to 5
while(number <= 10) {

  print(number)

  # increment number by 1
  number = number + 1

  # break if number is 6
  if (number == 6) {
    break
  }
```

```
}
```

**Output**

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

In this program, we have used a `break` statement inside the `while` loop, which breaks the loop as soon as the condition inside the `if` statement is evaluated to `TRUE`.

```
if (number == 6) {
  break
}
```

Hence, the loop terminates when the `number` variable equals to `6`. Therefore, only the numbers **1** to **5** are printed.

---

## Example 3: while Loop With next Statement

You can use the `next` statement in a `while` loop to skip an iteration even if the test condition is `TRUE`. For example,

```
number = 1

# while loop to print odd number between 1 to 10
while(number <= 10) {

  # skip iteration if number is even
  if (number %% 2 == 0) {
    number = number + 1
    next
  }
```

```
  # print number if odd
  print(number)

  # increment number by 1
  number = number + 1
}
```

**Output**

```
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
```

This program only prints the odd numbers in the range of **1** to **10**. To do this, we have used an `if` statement inside the `while` loop to check if `number` is divisible by **2**.

Here,

- if `number` is divisible by **2**, then its value is simply incremented by **1** and the iteration is skipped using the `next` statement.
- if `number` is not divisible by **2**, then the variable is printed and its value is incremented by **1**.