Identifiers

Definition:
Identifiers are names used to identify variables, functions, classes, and other objects in R. In R, an identifier:

- Must start with a letter or a dot (if not followed by a number).
- Is case-sensitive (e.g., `var1` and `Var1` are different).
- Can include letters, digits, dots, and underscores (but typically not spaces).

Example:

```
# Valid identifiers
x <- 10
var_1 <- "Hello, R!"
.dataItem <- 3.14

# Invalid identifiers (will throw an error)
# 1stVar <- 5      # Cannot start with a digit
```

Data Types

R supports several fundamental data types. Here are the primary ones:

Data Type	Description
Numeric	Represents numbers with decimal points.
Integer	Represents whole numbers. Integers are explicitly defined by appending an `L` to the number.
Character	Represents text or strings.
Logical	Represents Boolean values: `TRUE` or `FALSE`.

```r
# numeric
num <- 3.14159
class(num)  # Output: "numeric"

# integer
int <- 42L
class(int)  # Output: "integer"

# character (string)
char <- "R programming"
class(char)  # Output: "character"
```

Data Objects

R uses various objects to store and manipulate data. Here are the common data objects:

Vectors
A vector is a sequence of data elements of the same basic type. Vectors do not support additional dimensions or attributes such as names by default, but you can assign names later.

Example:

```
# Vector
vec <- c(1, 2, 3, 4, 5)
names(vec) <- c("a", "b", "c", "d", "e")
print(vec)

# Output:
#  a b c d e
```

The code creates a vector `vec` with numbers from 1 to 5 and then assigns custom names to each element.

Matrix

A matrix is a two-dimensional array arranged in rows and columns. You can customize the arrangement using `byrow` (to fill data row-wise) and assign row and column names with the `dimnames` parameter.

Example:

```
# Matrix
mat <- matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE,
              dimnames = list(c("Row1", "Row2", "Row3"),
                              c("Col1", "Col2", "Col3")))
print(mat)

# Output:
#      Col1 Col2 Col3
# Row1    1    2    3
```

Here, the matrix `mat` is created with values 1 through 9, filled row-wise. The `dimnames` argument assigns custom names to rows and columns.

Array An array extends matrices to more than two dimensions. You can specify multiple dimensions and provide names for each dimension with `dimnames`.

Example:

```
# Array
arr <- array(1:12, dim = c(3, 2, 2),
             dimnames = list(c("R1", "R2", "R3"),
                             c("C1", "C2"),
                             c("D1", "D2")))
print(arr)

# Output:
# , , D1
#     C1 C2
# R1   1  4
# R2   2  5
# R3   3  6
#
# , , D2
#     C1 C2
```

The array `arr` contains 12 elements structured as 3 rows, 2 columns, and 2 layers. The `dimnames` list labels each of these dimensions for easier interpretation.

Data Frame

A data frame is a table-like structure where each column can have different data types. You can also assign row names to describe each observation.

Example:

```
# Data Frame
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35),
  Member = c(TRUE, FALSE, TRUE),
  row.names = c("Obs1", "Obs2", "Obs3")
)
print(df)

# Output:
#        Name Age Member
# Obs1 Alice  25   TRUE
```

This data frame `df` holds columns of character, numeric, and logical types with designated row names. Each column can be of a different type, making data frames very versatile.

List

A list is a collection of objects that may contain different types of data (vectors, matrices, etc.). Each element can be named for clarity.

Example:

```
# List
my_list <- list(
  numbers = c(1, 2, 3),
  letters = c("a", "b", "c"),
  flag = TRUE,
  matrix_example = matrix(1:4, nrow = 2,
                          dimnames = list(c("Row1", "Row2"),
                                          c("Col1", "Col2")))
)
print(my_list)

# Output:
# $numbers
# [1] 1 2 3
#
# $letters
# [1] "a" "b" "c"
#
# $flag
# [1] TRUE
```

In this example, the list `my_list` contains a numeric vector, a character vector, a logical value, and a matrix with custom dimension names. This illustrates the flexibility of lists to hold various data types.

Operators

Operators are symbols that perform operations on variables and values. Below are categories of operators in R presented in tabular format.

Assignment Operators

Operator	Description	Example
`<-`	The primary assignment operator; assigns the value on the right to the left.	`x <- 10`
`->`	Rightward assignment operator; assigns the value on the left to the variable on the right.	`10 -> y`
`=`	Another assignment operator; similar in many cases to `<-`, but often used in function calls.	`z = 5`

Comparison Operators

Operator	Description	Example
`==`	Tests for equality.	`5 == 5 # returns TRUE`
`!=`	Tests for inequality.	`5 != 3 # returns TRUE`
`<`	Less than.	`3 < 5 # returns TRUE`
`>`	Greater than.	`10 > 2 # returns TRUE`
`<=`	Less than or equal to.	`4 <= 4 # returns TRUE`
`>=`	Greater than or equal to.	`7 >= 8 # returns FALSE`

Logical Operators

Operator	Description	Example
`&`	Element-wise logical AND	`c(TRUE, FALSE) & c(TRUE, TRUE)` → `c(TRUE, FALSE)`
`|`	Element-wise logical OR	`c(TRUE, FALSE) | c(FALSE, TRUE)` → `c(TRUE, TRUE)`
`!`	Logical NOT (negation)	`!TRUE` → `FALSE`
`&&`	Logical AND for **first element only**; used in conditionals	`TRUE && FALSE` → `FALSE`
`||`	Logical OR for **first element only**; used in conditionals	`TRUE || FALSE` → `TRUE`

Miscellaneous Operators

Operator	Description	Example
`%in%`	Checks if a value exists in a vector or list.	`3 %in% c(1, 2, 3, 4) # returns TRUE`
`:`	Sequence operator; creates a sequence of numbers.	`1:5 # returns c(1, 2, 3, 4, 5)`

Conditional Statements in R

Conditional statements allow you to control the flow of your program based on **logical conditions** — running different blocks of code depending on whether a condition is `TRUE` or `FALSE`.

`if` Statement

Definition:
The `if` statement evaluates a condition. If the condition is `TRUE`, the associated code block is executed. If it's `FALSE`, nothing happens (unless there's an `else` or `else if`).

Syntax:

```
if (condition) {
  # code to run if condition is TRUE
}
```

Example:

```
x <- 10

if (x > 5) {
  print("x is greater than 5")
}

# Output:
```

`if...else` Statement

Definition:
Used when you want to specify an **alternative block of code** that should run if the condition is `FALSE`.

Syntax:

```
if (condition) {
  # code if condition is TRUE
} else {
  # code if condition is FALSE
}
```

Example:

```
x <- 3

if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

`ifelse()` Function

Definition:
The `ifelse()` function is a **vectorized** conditional function. It checks a condition for **each element of a vector** and returns one value if `TRUE`, another if `FALSE`.

Syntax:

```
ifelse(test_expression, value_if_true, value_if_false)
```

Example:

```
x <- 1:10
result <- ifelse(x %% 2 == 0, "even", "odd")
print(result)

# Output:
# [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

Useful when labeling data, classifying values, or applying different operations in bulk.

Loops in R

Loops allow you to **automate repetitive tasks** by executing a block of code multiple times, either over a **sequence of elements** or **until a condition is met**.

`for` Loop

Definition:
The `for` loop is used to iterate over each element in a **vector, list, or sequence**, executing a block of code for every element.

Syntax:

```
for (variable in sequence) {
  # Code block to execute
}
```

Example:

```
# Print each number in a vector
numbers <- c(1, 2, 3, 4, 5)

for (num in numbers) {
  print(num)
}

# Output:
# [1] 1
# [1] 2
# [1] 3
```

Use Case:

- Iterating over rows in a dataset
- Applying transformations to each item in a vector or list
- Generating reports or summaries per group

`while` Loop

Definition:
The `while` loop runs a block of code **as long as a condition remains** `TRUE`.

Syntax:

```
while (condition) {
  # Code block to execute
}
```

Example:

```
# Print numbers until counter exceeds 5
counter <- 1

while (counter <= 5) {
  print(counter)
  counter <- counter + 1
}

# Output:
# [1] 1
# [1] 2
# [1] 3
```

Use Case:

- Waiting for a condition to be satisfied before proceeding
- Reading files line by line until end
- Polling a process or server status

`repeat` Loop

Definition:
The `repeat` loop runs **indefinitely** until a `break` statement is encountered. It's useful when the number of iterations is **unknown** at the start.

Syntax:

```
repeat {
  # Code block to execute
  if (exit_condition) {
    break
  }
```

Example:

```
# Print numbers until counter exceeds 5 using repeat
counter <- 1

repeat {
  print(counter)
  counter <- counter + 1
  if (counter > 5) {
    break  # Exit loop
  }
}

# Output:
# [1] 1
# [1] 2
```

Use Case:

- Retry mechanisms (e.g., retrying a connection until successful)
- Creating user input loops
- Waiting for an event to occur, then breaking
-

`break` and `next` in R Loops

- These are **loop control statements** in R:

Statement	Description
`break`	Immediately exits the loop, stopping further iterations.
`next`	Skips the **current iteration** and moves to the **next one** without executing the rest of the loop body.

`break` – Exit the Loop Early

Definition:
Use `break` when you want to **terminate the loop early**, usually when a specific condition is met.

Example: Stop loop when number equals 3

```
for (i in 1:5) {
  if (i == 3) {
    break
  }
  print(i)
}

# Output:
# [1] 1
```

The loop stops completely when `i == 3`. Anything after that is not executed.

`next` — Skip Current Iteration

Definition:
Use `next` when you want to **skip certain values** or iterations **without stopping the loop**.

Example: Skip number 3

```
for (i in 1:5) {
  if (i == 3) {
    next
  }
  print(i)
}

# Output:
# [1] 1
# [1] 2
```

When `i == 3`, the loop **skips the print statement** and moves to the next value of `i`.

Use Case	Use `break`	Use `next`
Stop processing once a match is found	✅	❌
Skip certain values (e.g., NA, negative)	❌	✅
Infinite loop with controlled exit	✅	❌

Functions in R

Definition:
A **function** in R is a block of reusable code designed to perform a specific task. It takes input (arguments), processes them, and returns an output.

Syntax of a Function

```
function_name <- function(arg1, arg2, ...) {
  # Code block (body of the function)
  return(result)
}
```

- `function_name`: The name of your function
- `arg1`, `arg2`, ...: Arguments (inputs)
- `return()`: Returns the output (optional; if not used, R returns the last evaluated expression)

Example: Define and call a function

```
# Define a simple function that adds two numbers
add_numbers <- function(a, b) {
  return(a + b)
}

# Call the function
result <- add_numbers(10, 20)
```

Argument Matching in R

When calling a function, R matches the values you provide with the function's parameters using the following strategies:

1. Positional Matching

- Arguments are matched in the **order they are provided**.
- The first value goes to the first parameter, the second to the second, and so on.

```
add_numbers(10, 20)  # 10 → a, 20 → b
```

Simple and readable when calling short functions with obvious order.

2. Named Matching

- Arguments are matched using explicit parameter names.
- You can pass arguments in any order.

```
add_numbers(b = 20, a = 10)  # Order doesn't matter
```

Increases clarity, especially in functions with many arguments.

3. Partial Matching

R allows matching using abbreviated names, as long as they are unambiguous.

Example:

```
add <- function(first, second) {
  return(first + second)
}

add(fir = 5, sec = 10)  # R matches fir → first, sec → second
```

Partial matching can make code less readable and error-prone, especially in large or shared codebases.

Use Case Example

Let’s define a more realistic function with multiple arguments:

```
create_user <- function(name, age, is_member = FALSE) {
  if (is_member) {
    paste(name, "is a member and is", age, "years old.")
  } else {
    paste(name, "is not a member and is", age, "years old.")
  }
}

# Named matching (clear and flexible)
print(create_user(age = 30, name = "Alice", is_member = TRUE))
```