Programming Essential in R

Section-2

Learning Objectives

By the end of this lesson, you will be able to:

- Explain and use conditional statements
- Create and use different loops in R
- Understand the loop controls in R
- Define and use functions in R
- Use other important Built-in functions in R

Conditional Statements in R

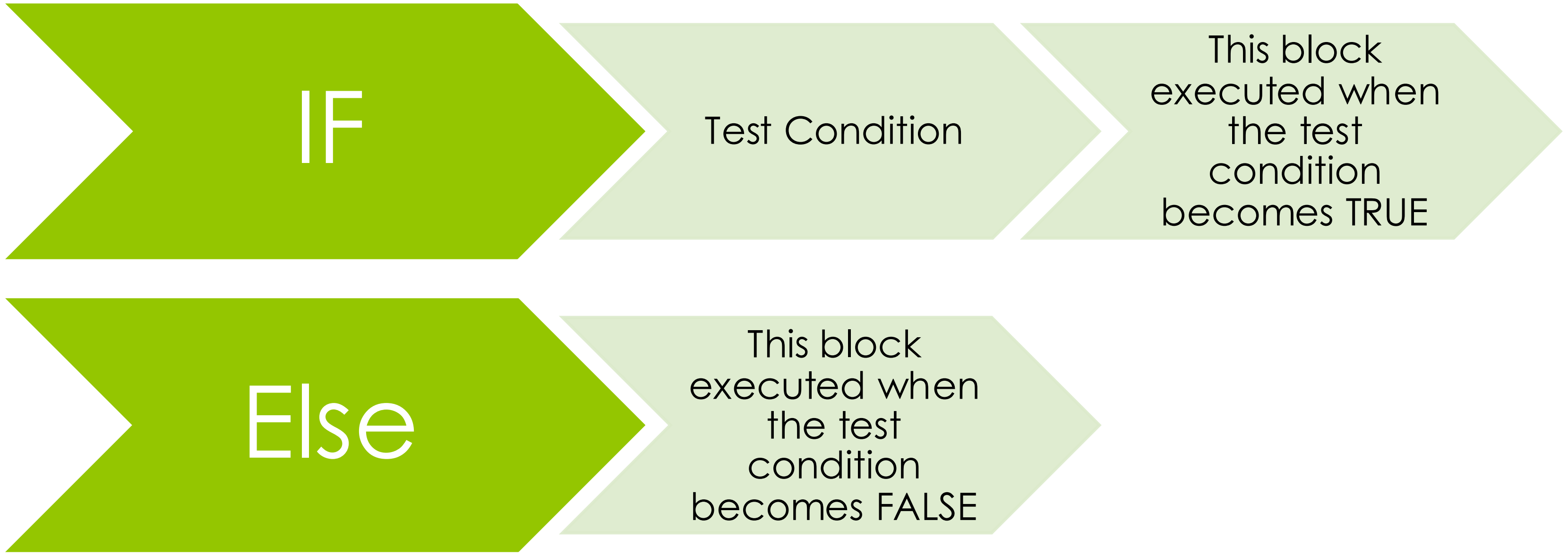
The use of **conditional statements** in programming languages is fundamental for **decision-making** and **logical thinking**.

- IF
- IF ELSE
- IF ELSE IF ……….

- Each condition returns either True or False.

IF ELSE

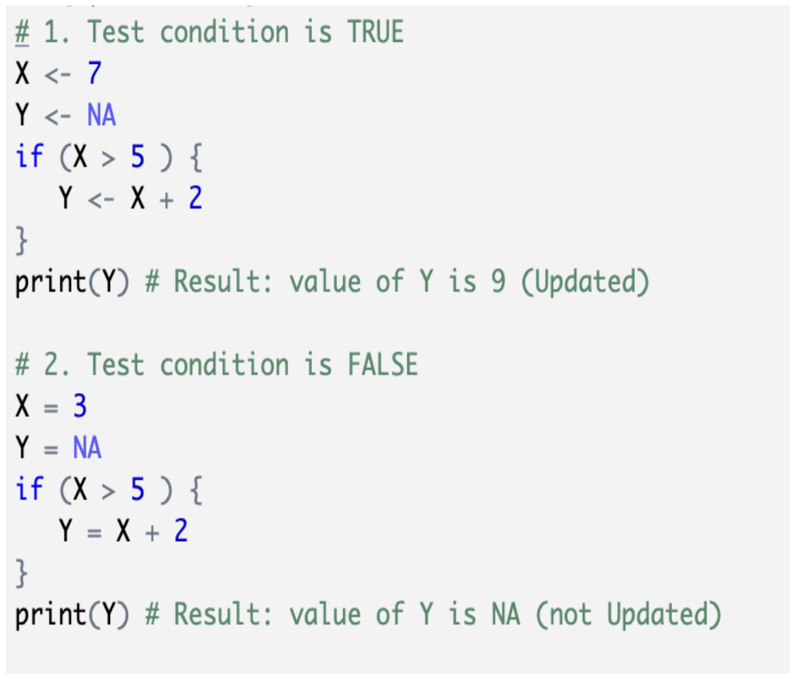
Uses: when you need to make a decision between two possibilities in a program


IF
Test Condition
This block executed when the test condition becomes TRUE
Else
This block executed when the test condition becomes FALSE

IF:Syntax

- **If**

```
if (Test Condition) {
        Expression to execute when the test condition TRUE
}
```

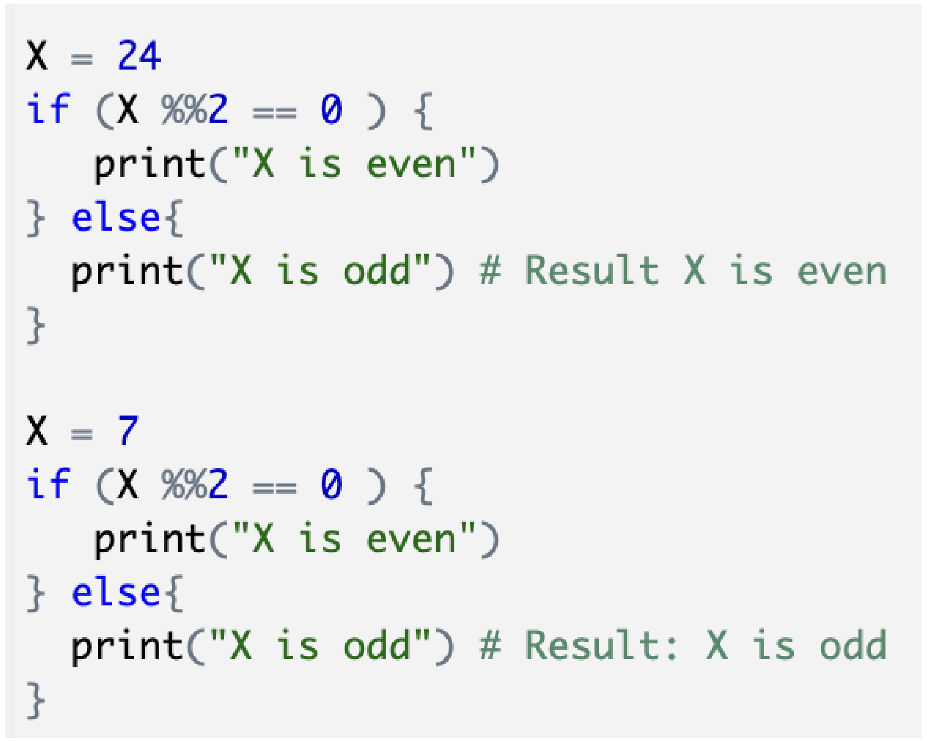


```
# 1. Test condition is TRUE
X <- 7
Y <- NA
if (X > 5 ) {
   Y <- X + 2
}
print(Y) # Result: value of Y is 9 (Updated)

# 2. Test condition is FALSE
X = 3
Y = NA
if (X > 5 ) {
   Y = X + 2
}
print(Y) # Result: value of Y is NA (not Updated)
```

IF ELSE: Syntax

- **if else**

```
if (Test Condition) {
        Expression to execute when the test condition TRUE
} else {
        Expression to execute when the test condition FALSE
}
```



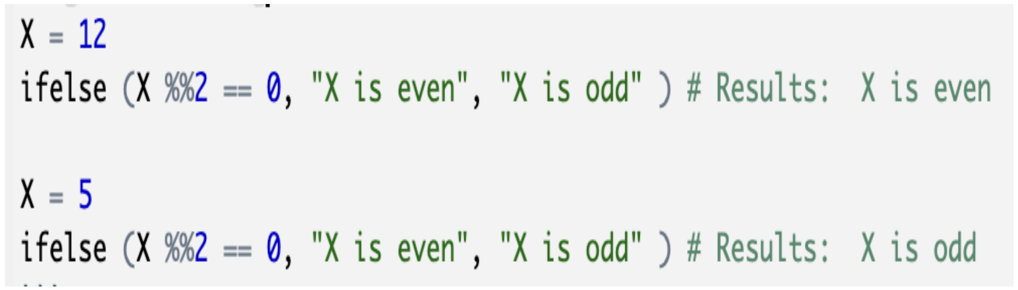
```
X = 24
if (X %%2 == 0 ) {
   print("X is even")
} else{
  print("X is odd") # Result X is even
}

X = 7
if (X %%2 == 0 ) {
   print("X is even")
} else{
  print("X is odd") # Result: X is odd
}
```

ifelse is equivalent to if else

Syntax:

ifelse(Test Condition, Expression-1, Expression-2)

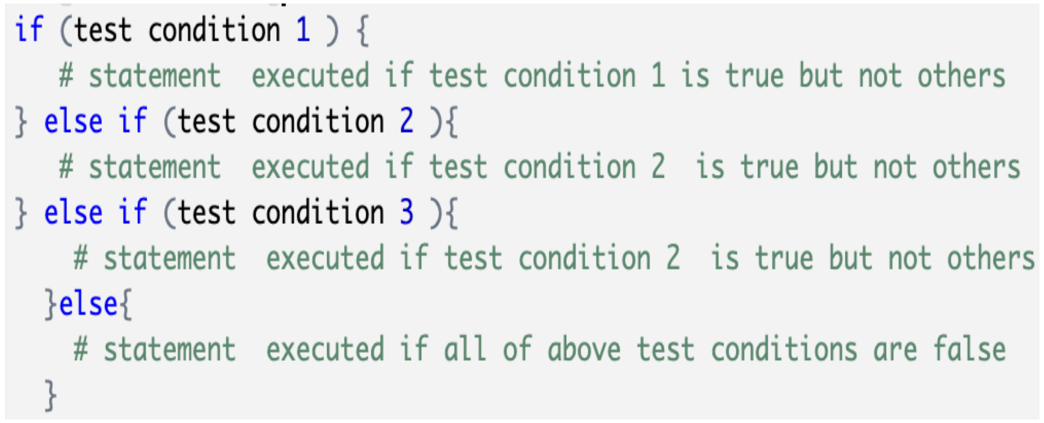
- *Expression-1 executed if the test condition TRUE,*
- *Expression-2 executed if the test condition FALSE*



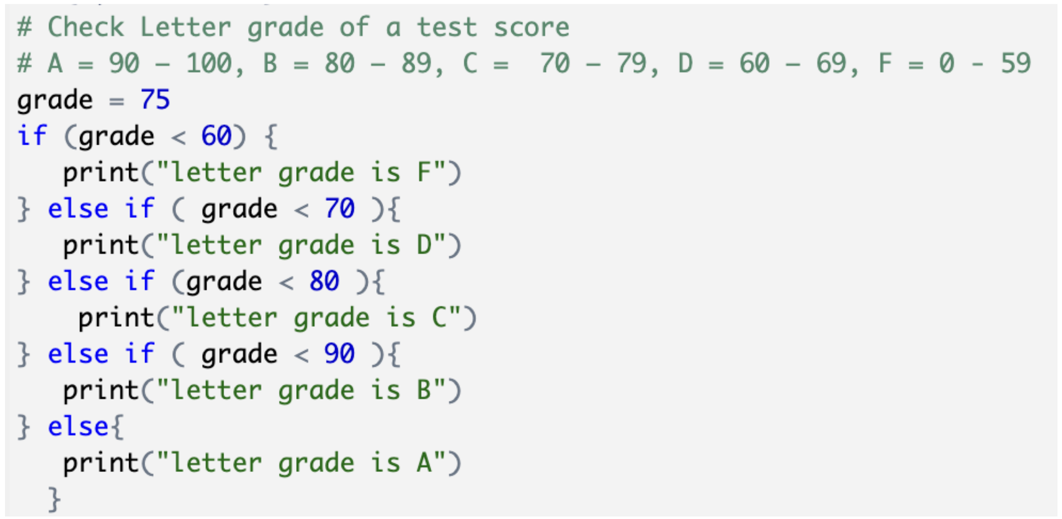
```
X = 12
ifelse (X %%2 == 0, "X is even", "X is odd" ) # Results:  X is even

X = 5
ifelse (X %%2 == 0, "X is even", "X is odd" ) # Results:  X is odd
```

IF ELSE IF… (Conditional Chain)

- This is a **linear sequence** of conditions.
- It checks multiple conditions **one by one** until it finds a true condition.



```
if (test condition 1 ) {
   # statement  executed if test condition 1 is true but not others
} else if (test condition 2 ){
   # statement  executed if test condition 2  is true but not others
} else if (test condition 3 ){
    # statement  executed if test condition 2  is true but not others
  }else{
    # statement  executed if all of above test conditions are false
  }
```



```
# Check Letter grade of a test score
# A = 90 - 100, B = 80 - 89, C =  70 - 79, D = 60 - 69, F = 0 - 59
grade = 75
if (grade < 60) {
   print("letter grade is F")
} else if ( grade < 70 ){
   print("letter grade is D")
} else if (grade < 80 ){
    print("letter grade is C")
} else if ( grade < 90 ){
   print("letter grade is B")
} else{
   print("letter grade is A")
  }
```

LOOPS

- Uses: repetitive tasks, automate processes, and enhance code efficiency
- Types:
  - For
  - While
  - Repeat

For Loop

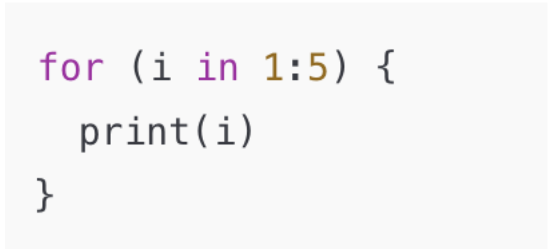
- The mostly commonly used loop structure to compute repeated tasks up to the last entry of a sequence
- Uses: when the number of iterations is **known**.

Syntax:

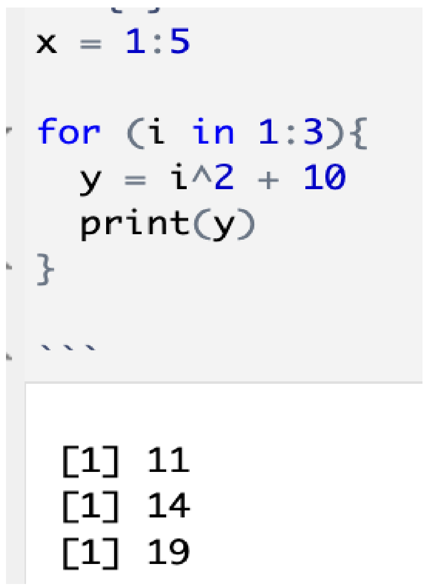
for (variable in sequence) {

# Code to execute

}



```
for (i in 1:5) {
  print(i)
}
```

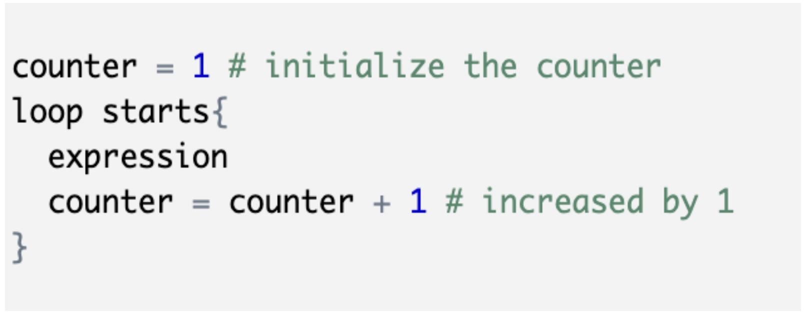


```
x = 1:5

for (i in 1:3){
  y = i^2 + 10
  print(y)
}

```

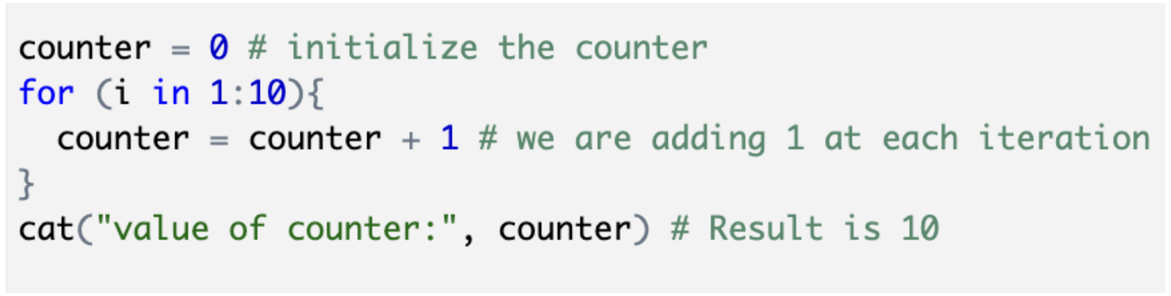
```
[1] 11
[1] 14
[1] 19
```

Counter in programming language

- a variable used in programming to **keep track of iterations**
- typically increases or decreases by a fixed value



```
counter = 1 # initialize the counter
loop starts{
  expression
  counter = counter + 1 # increased by 1
}
```

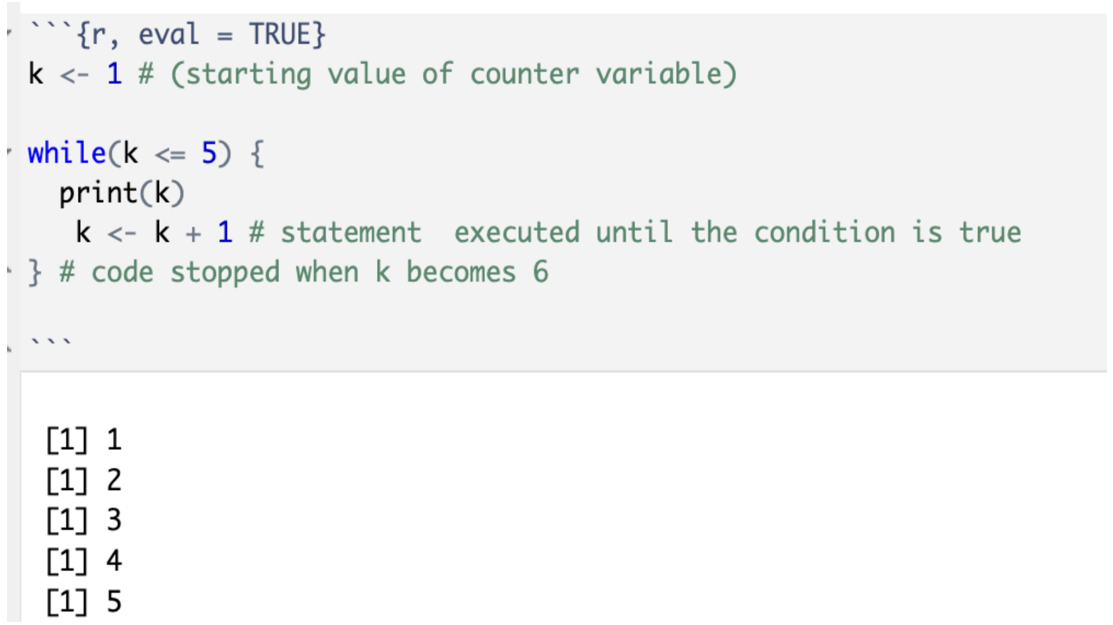


```
counter = 0 # initialize the counter
for (i in 1:10){
  counter = counter + 1 # we are adding 1 at each iteration
}
cat("value of counter:", counter) # Result is 10
```

While Loop

- **Uses:** when the number of iterations is **unknown,** and it depends on a condition.
- Code inside the loop executed until the **condition is TRUE.**

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



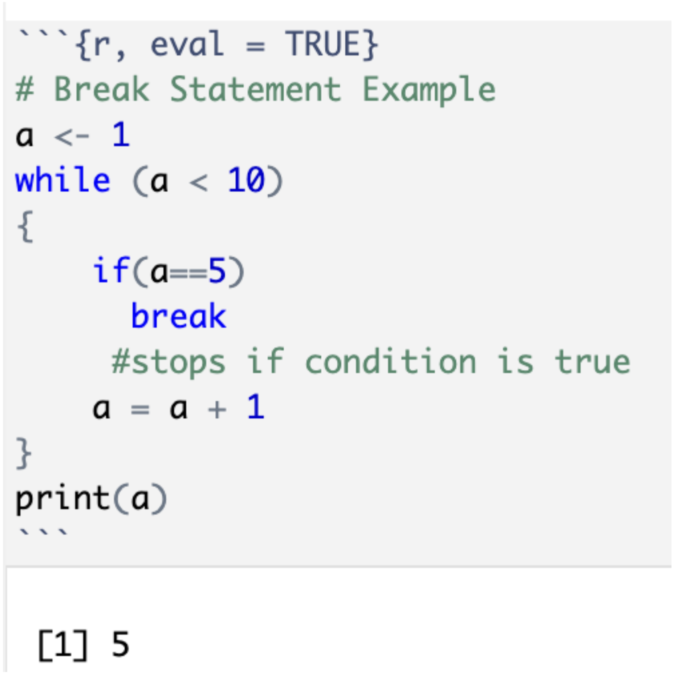
~~~
```{r, eval = TRUE}
k <- 1 # (starting value of counter variable)

while(k <= 5) {
  print(k)
   k <- k + 1 # statement  executed until the condition is true
} # code stopped when k becomes 6

```
~~~

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

Loop Control Statements

- break:

Uses: inside a loop (repeat, for, while) to stop the iterations.



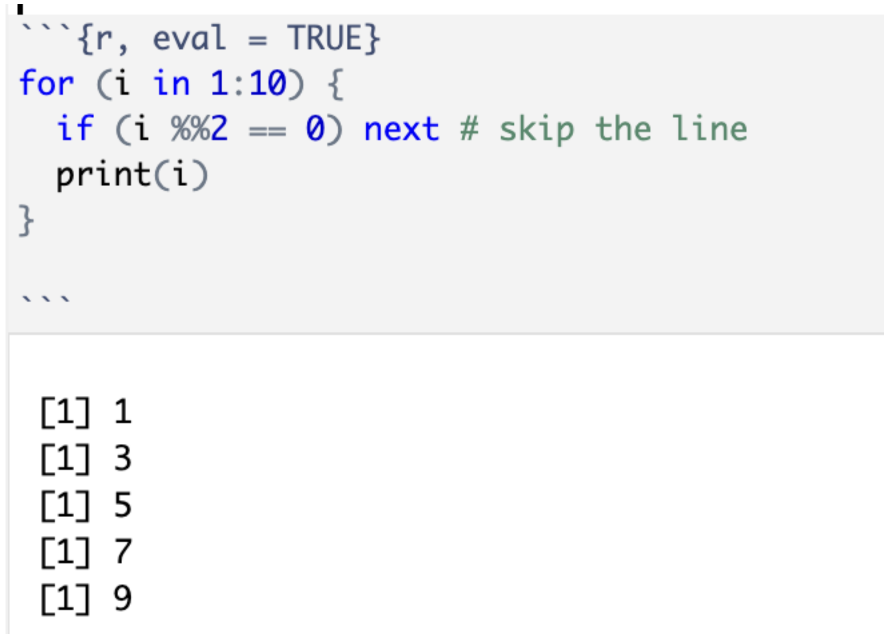
~~~
```{r, eval = TRUE}
# Break Statement Example
a <- 1
while (a < 10)
{
    if(a==5)
      break
     #stops if condition is true
    a = a + 1
}
print(a)
```
~~~

```
[1] 5
```

Loop Control Statements

- next:

Uses: skip the current iteration in the loop and move to the next iteration without exiting.



~~~
```{r, eval = TRUE}
for (i in 1:10) {
  if (i %%2 == 0) next # skip the line
  print(i)
}

```
~~~

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

Repeat Loop

Uses: when you want to **force** the loop to run until a break condition is met.

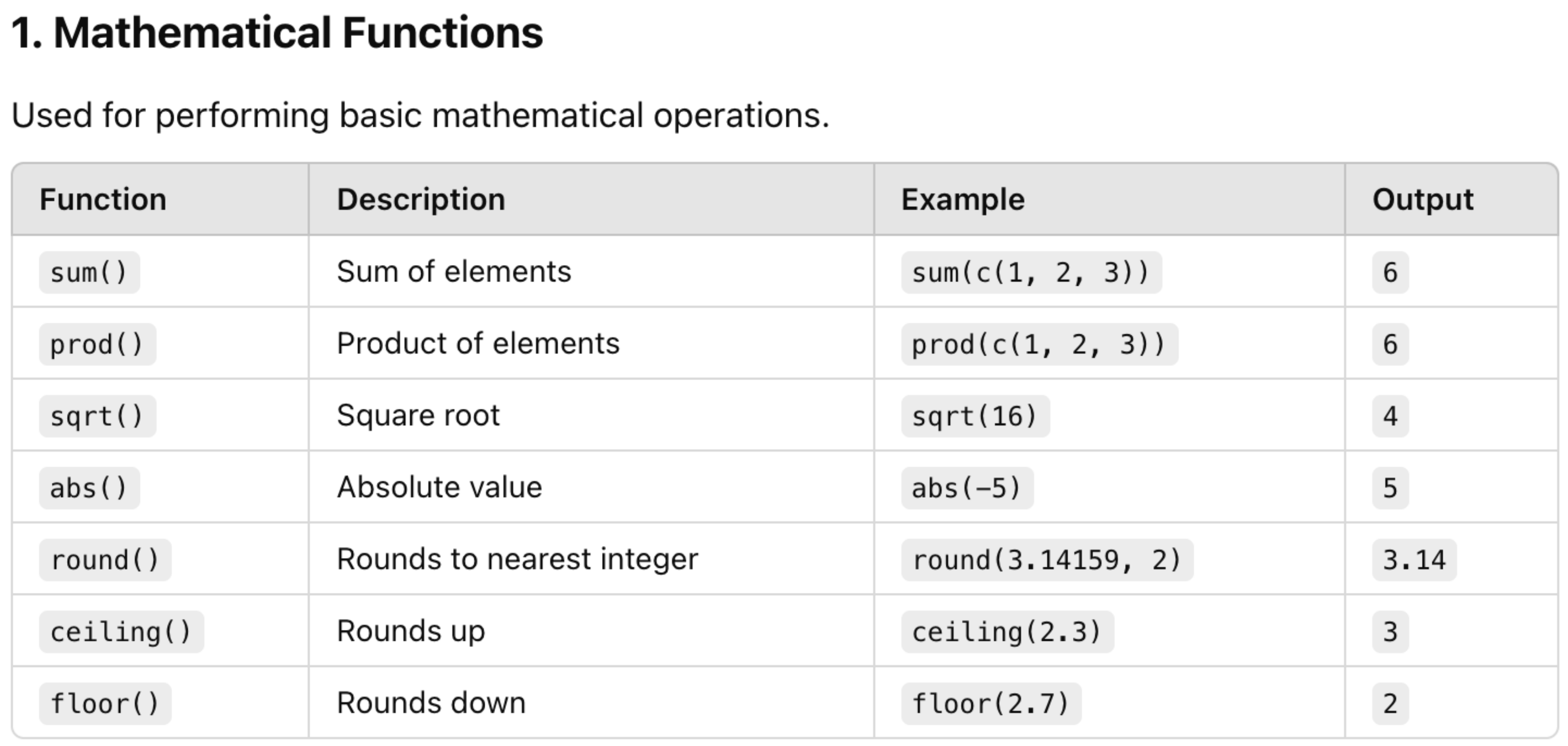
```
repeat {
  # Code to execute
  if (condition) {
    break   # Exit loop
  }
}
```

~~~
```{r, eval=TRUE}
k = 1
sumk = k
# Repeat loop syntax
repeat {
  k = k + 1
   if(k > 10) {
      break # loop stopped if condition is true.
   }
  sumk = sumk + k
  print(paste("Sum up to", k, 'is', sumk, sep = " "))
}
```
~~~

```
[1] "Sum up to 2 is 3"
[1] "Sum up to 3 is 6"
[1] "Sum up to 4 is 10"
[1] "Sum up to 5 is 15"
[1] "Sum up to 6 is 21"
[1] "Sum up to 7 is 28"
[1] "Sum up to 8 is 36"
[1] "Sum up to 9 is 45"
[1] "Sum up to 10 is 55"
```

Functions in R

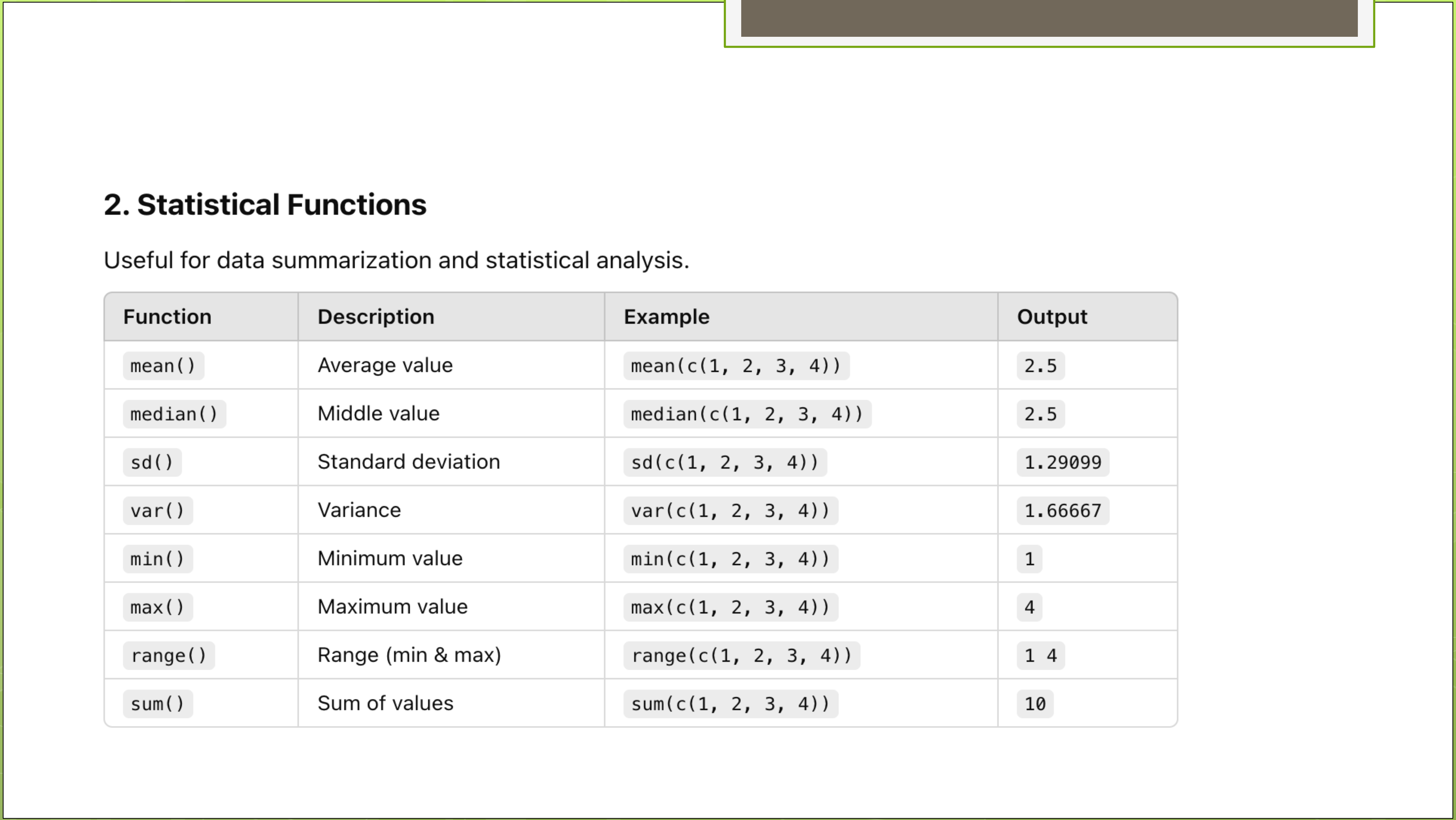
There are many built in functions in R such as



1. Mathematical Functions

Used for performing basic mathematical operations.

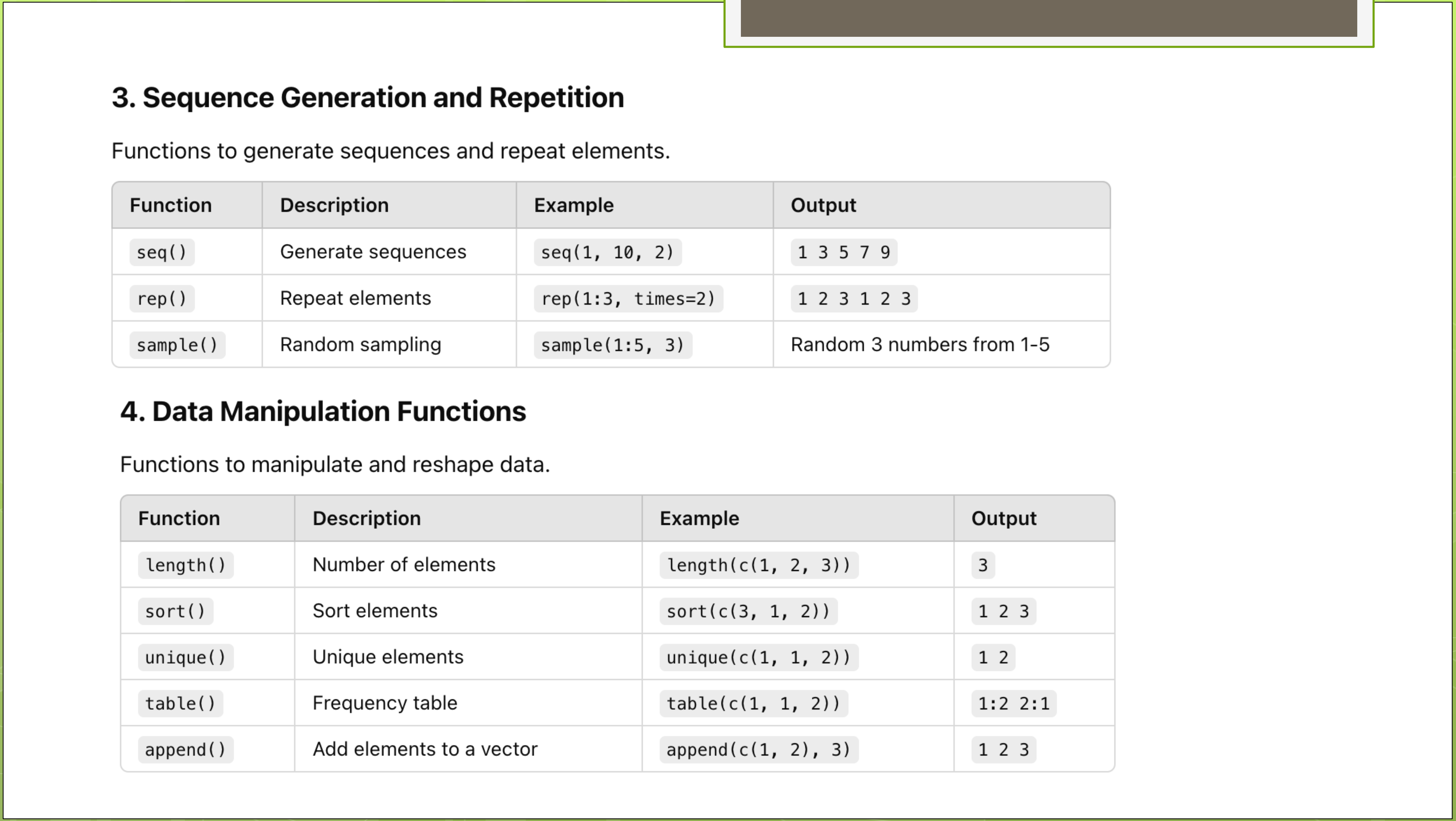
Function	Description	Example	Output
`sum()`	Sum of elements	`sum(c(1, 2, 3))`	`6`
`prod()`	Product of elements	`prod(c(1, 2, 3))`	`6`
`sqrt()`	Square root	`sqrt(16)`	`4`
`abs()`	Absolute value	`abs(-5)`	`5`
`round()`	Rounds to nearest integer	`round(3.14159, 2)`	`3.14`
`ceiling()`	Rounds up	`ceiling(2.3)`	`3`
`floor()`	Rounds down	`floor(2.7)`	`2`



2. Statistical Functions

Useful for data summarization and statistical analysis.

Function	Description	Example	Output
`mean()`	Average value	`mean(c(1, 2, 3, 4))`	`2.5`
`median()`	Middle value	`median(c(1, 2, 3, 4))`	`2.5`
`sd()`	Standard deviation	`sd(c(1, 2, 3, 4))`	`1.29099`
`var()`	Variance	`var(c(1, 2, 3, 4))`	`1.66667`
`min()`	Minimum value	`min(c(1, 2, 3, 4))`	`1`
`max()`	Maximum value	`max(c(1, 2, 3, 4))`	`4`
`range()`	Range (min & max)	`range(c(1, 2, 3, 4))`	`1 4`
`sum()`	Sum of values	`sum(c(1, 2, 3, 4))`	`10`



3. Sequence Generation and Repetition

Functions to generate sequences and repeat elements.

Function	Description	Example	Output
`seq()`	Generate sequences	`seq(1, 10, 2)`	`1 3 5 7 9`
`rep()`	Repeat elements	`rep(1:3, times=2)`	`1 2 3 1 2 3`
`sample()`	Random sampling	`sample(1:5, 3)`	Random 3 numbers from 1-5

4. Data Manipulation Functions

Functions to manipulate and reshape data.

Function	Description	Example	Output
`length()`	Number of elements	`length(c(1, 2, 3))`	`3`
`sort()`	Sort elements	`sort(c(3, 1, 2))`	`1 2 3`
`unique()`	Unique elements	`unique(c(1, 1, 2))`	`1 2`
`table()`	Frequency table	`table(c(1, 1, 2))`	`1:2 2:1`
`append()`	Add elements to a vector	`append(c(1, 2), 3)`	`1 2 3`

5. Character/String Functions

Functions for text processing.

Function	Description	Example	Output
`paste()`	Concatenate strings	`paste('Hello', 'R')`	`"Hello R"`
`substr()`	Extract substring	`substr('Hello', 1, 3)`	`"Hel"`
`toupper()`	Convert to uppercase	`toupper('hello')`	`"HELLO"`
`tolower()`	Convert to lowercase	`tolower('HELLO')`	`"hello"`
`nchar()`	Number of characters	`nchar('Hello')`	`5`

6. Logical Functions

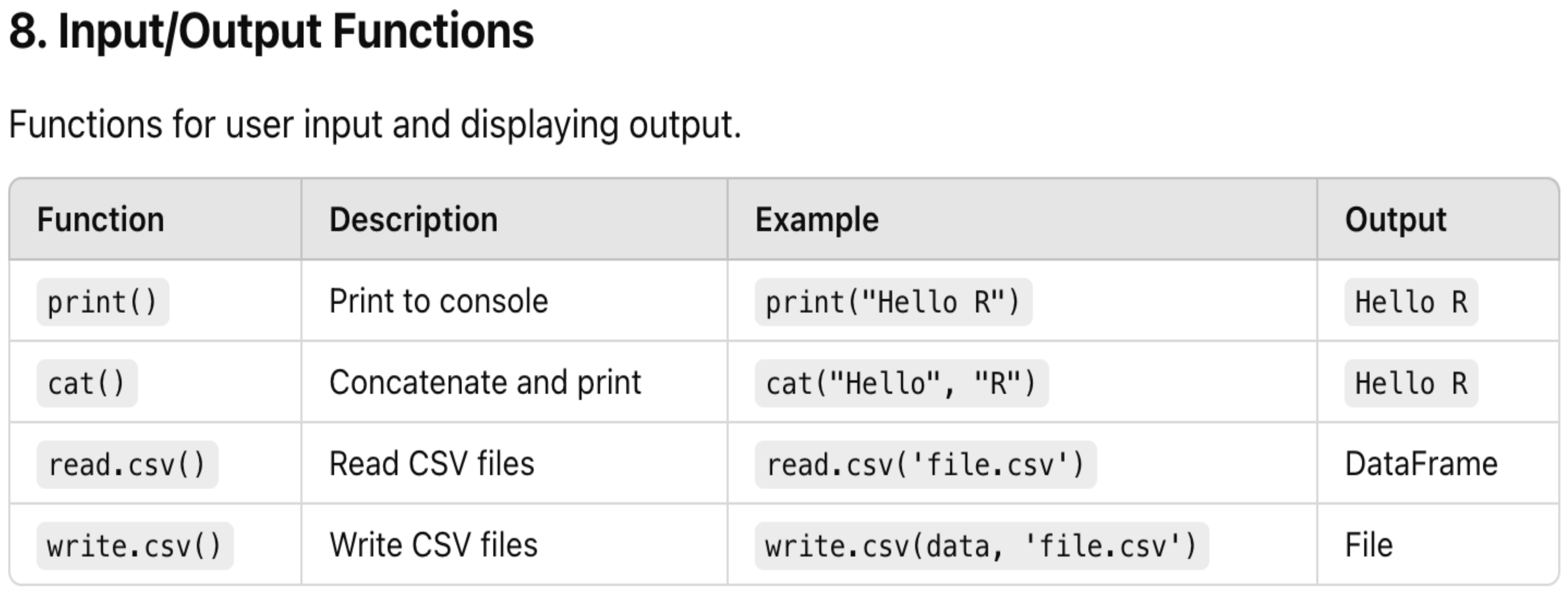
Functions that return logical values.

Function	Description	Example	Output
`all()`	Checks if all are TRUE	`all(c(TRUE, FALSE))`	`FALSE`
`any()`	Checks if any are TRUE	`any(c(TRUE, FALSE))`	`TRUE`
`which()`	Returns index of TRUE	`which(c(FALSE, TRUE))`	`2`
`is.na()`	Check for NA values	`is.na(c(1, NA, 2))`	`FALSE TRUE FALSE`

7. Apply Family Functions

Efficient alternatives to loops for applying functions.

Function	Description	Example	Output
`apply()`	Apply a function over rows/columns of a matrix	`apply(matrix(1:4, 2), 1, sum)`	`3 7`
`lapply()`	Apply a function over a list and return a list	`lapply(1:3, sqrt)`	`1 1.414 1.732`
`sapply()`	Same as `lapply()` but returns a vector	`sapply(1:3, sqrt)`	`1 1.414 1.732`
`tapply()`	Apply function over subsets	`tapply(1:6, c(1,1,2,2,3,3), sum)`	`3 7 11`



8. Input/Output Functions

Functions for user input and displaying output.

Function	Description	Example	Output
`print()`	Print to console	`print("Hello R")`	`Hello R`
`cat()`	Concatenate and print	`cat("Hello", "R")`	`Hello R`
`read.csv()`	Read CSV files	`read.csv('file.csv')`	DataFrame
`write.csv()`	Write CSV files	`write.csv(data, 'file.csv')`	File

User defined functions

- While R offers many powerful **built-in functions**, they can't cover every possible task.
- Custom functions allow us to design solutions for **specific** to to our unique needs, models, or workflows.

Defining Function in R

Use the **function()** keyword to define a function in R.

Key Components:

- Function Name: Identifier for the function.
- Arguments/Parameters: Input values for processing.
- Function Body: The logic/code to perform tasks.
- Return Statement: Outputs the result (optional but recommended).

Syntax:

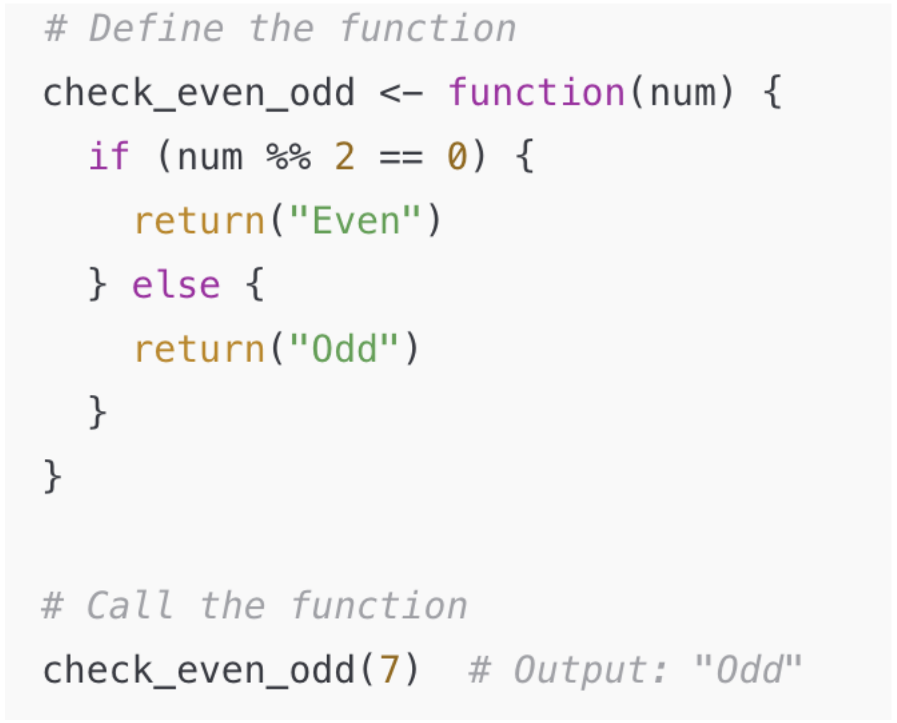
```
function_name <- function(arg1, arg2, ...) {
  # Body of the function (code to execute)
  return(result)  # Optional: Returns a value
}
```

Function calling:

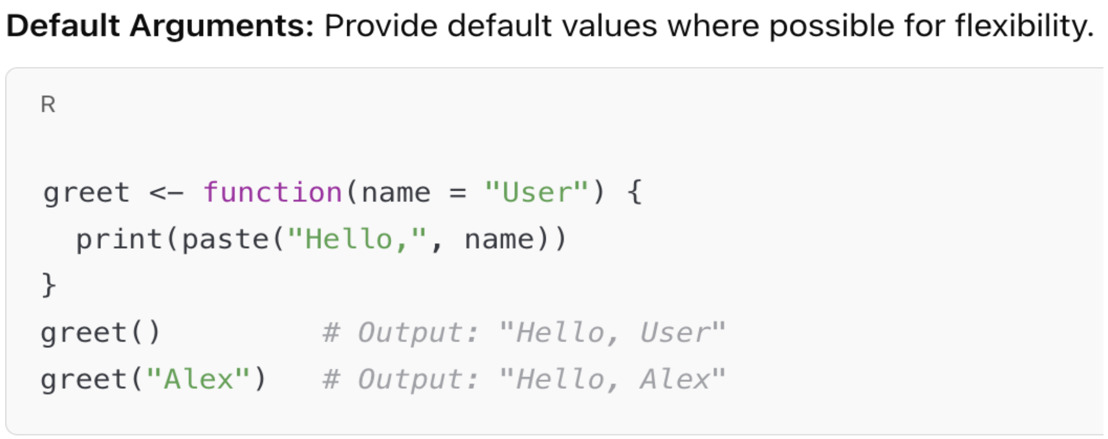
```
function_name(arg1_value, arg2_value, ...)
```

Function Examples

Function check even numbers



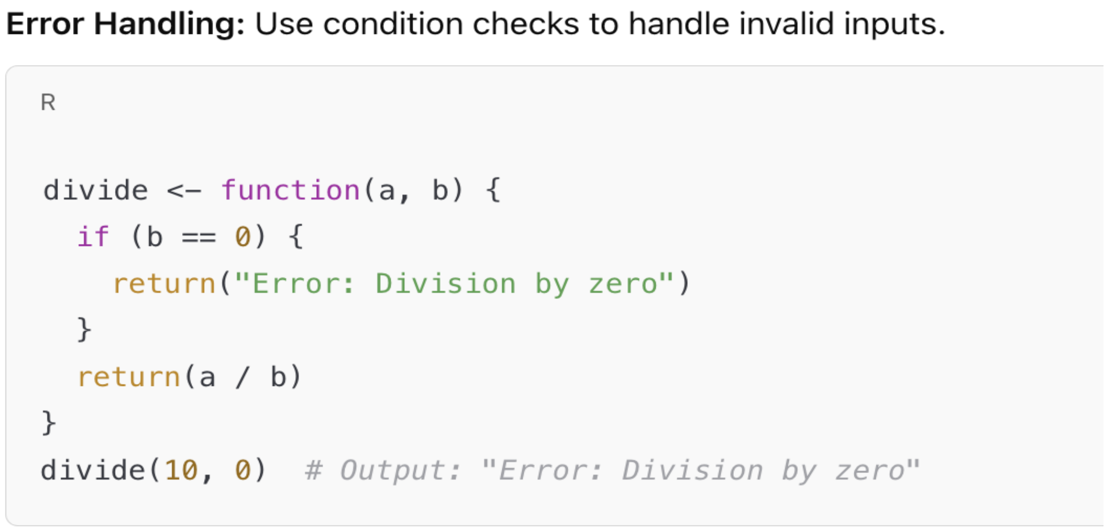
```r
# Define the function
check_even_odd <- function(num) {
  if (num %% 2 == 0) {
    return("Even")
  } else {
    return("Odd")
  }
}

# Call the function
check_even_odd(7)  # Output: "Odd"
```



Default Arguments: Provide default values where possible for flexibility.

R

```r
greet <- function(name = "User") {
  print(paste("Hello,", name))
}
greet()          # Output: "Hello, User"
greet("Alex")    # Output: "Hello, Alex"
```



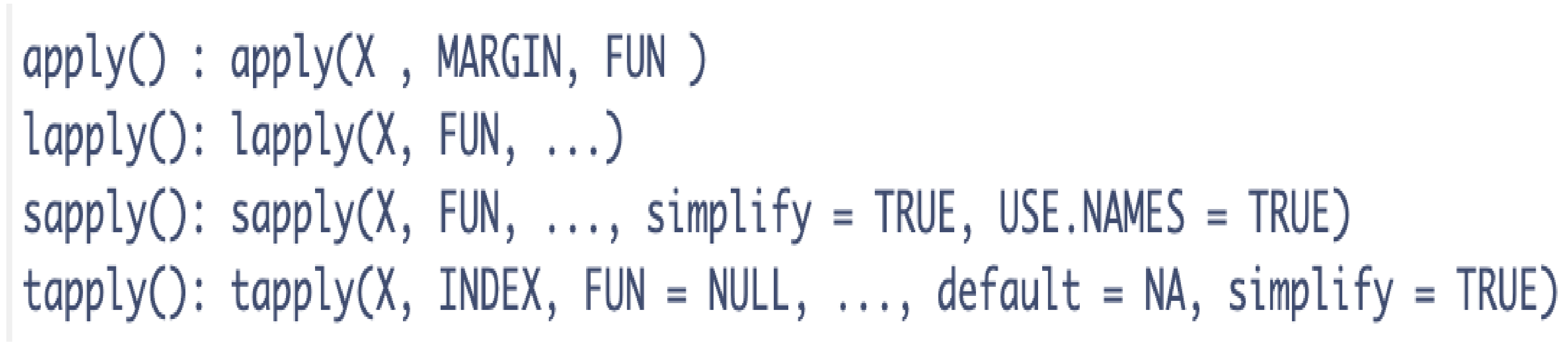
Error Handling: Use condition checks to handle invalid inputs.

R

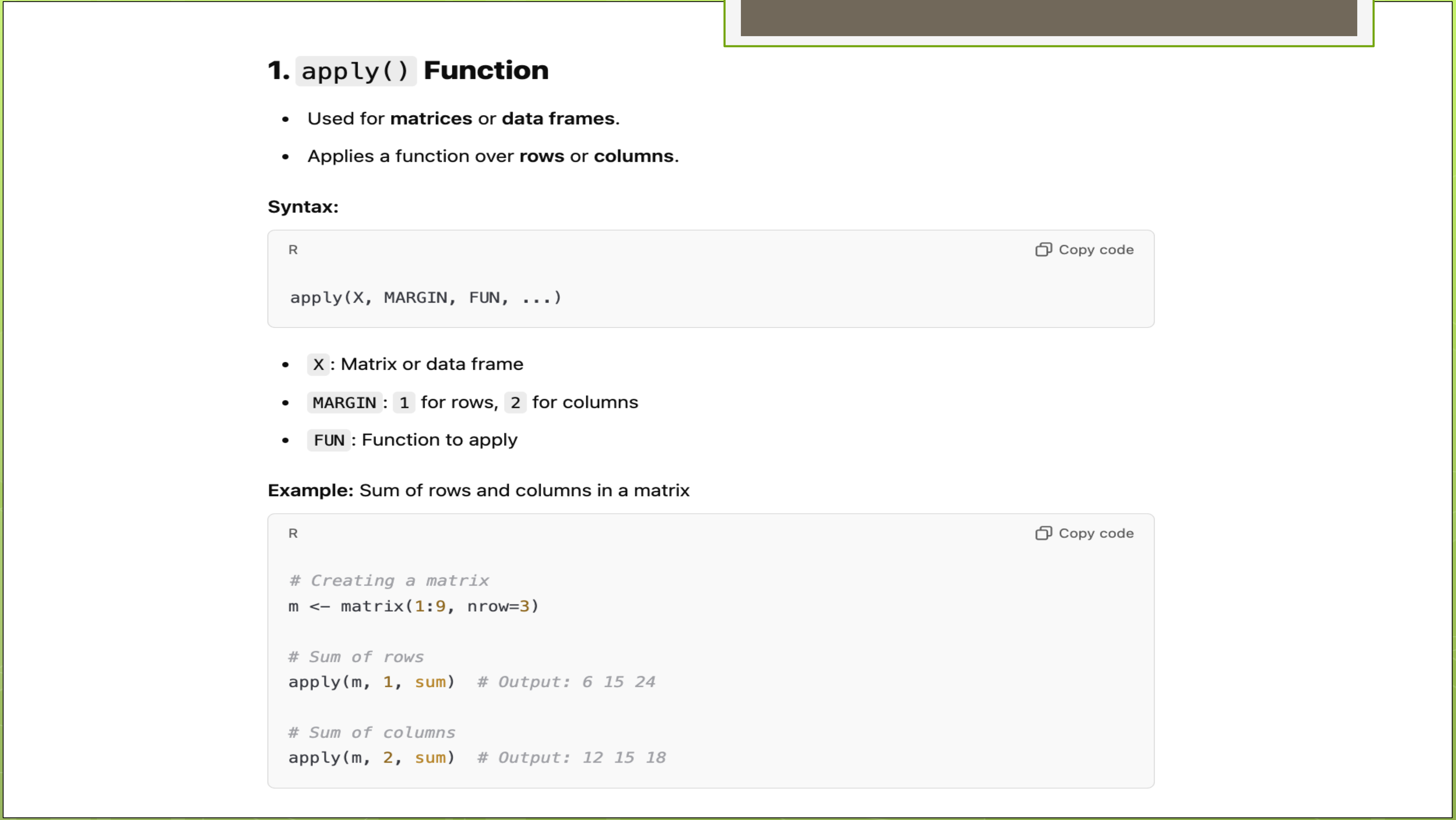
```r
divide <- function(a, b) {
  if (b == 0) {
    return("Error: Division by zero")
  }
  return(a / b)
}
divide(10, 0)  # Output: "Error: Division by zero"
```

Apply Family Functions

- **Uses:** to manipulate the data to avoid explicit use of loop constructs.
- **Input:** list, matrix or array and
- apply a named function with one or several optional arguments.
- **Most common functions:**



```
apply() : apply(X , MARGIN, FUN )
lapply(): lapply(X, FUN, ...)
sapply(): sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
tapply(): tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```



1. `apply()` Function

- Used for **matrices** or **data frames**.
- Applies a function over **rows** or **columns**.

Syntax:

R

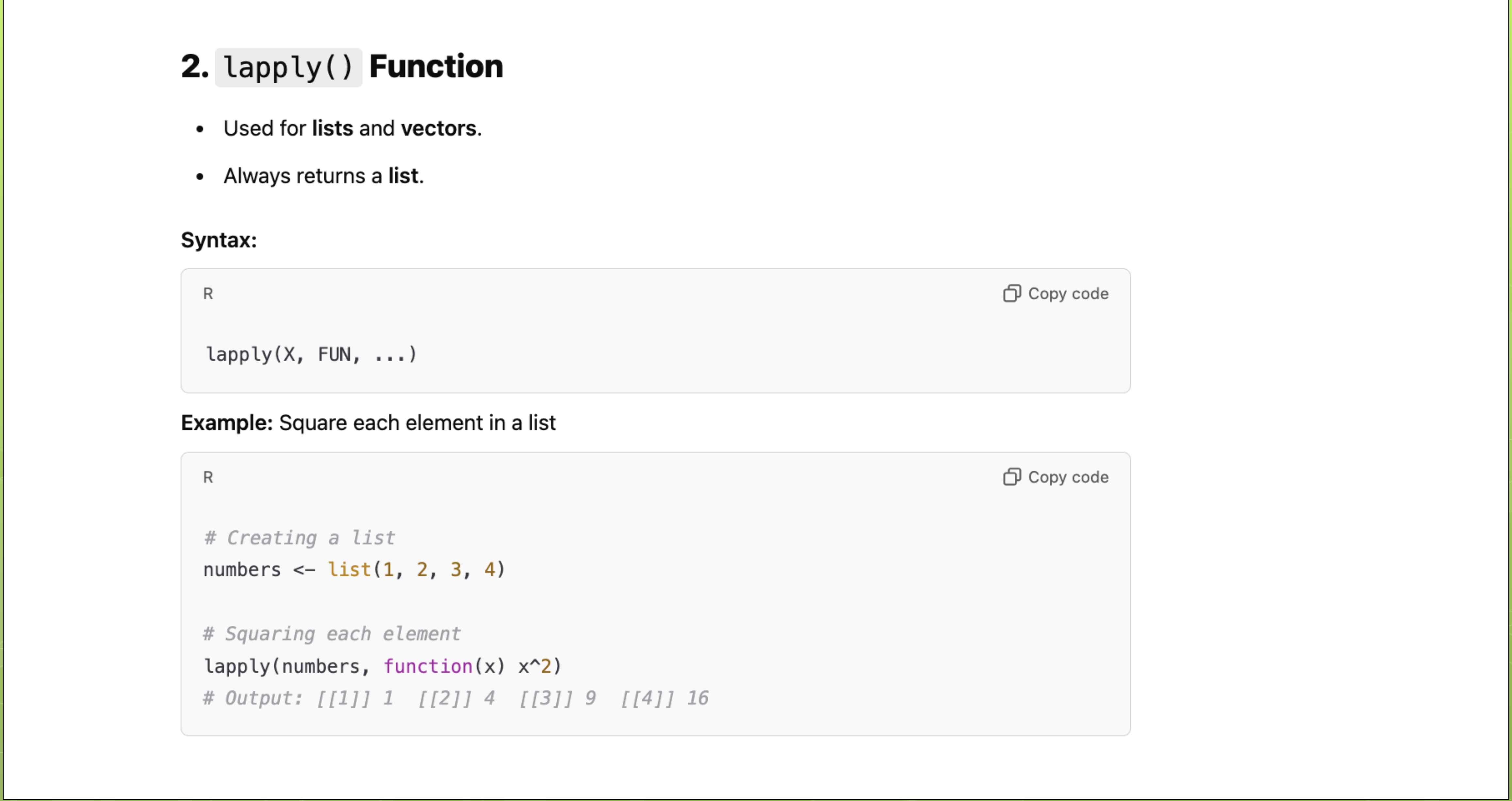
```r
apply(X, MARGIN, FUN, ...)
```

- `X` : Matrix or data frame
- `MARGIN` : `1` for rows, `2` for columns
- `FUN` : Function to apply

Example: Sum of rows and columns in a matrix

R

```r
# Creating a matrix
m <- matrix(1:9, nrow=3)

# Sum of rows
apply(m, 1, sum)  # Output: 6 15 24

# Sum of columns
apply(m, 2, sum)  # Output: 12 15 18
```



2. `lapply()` Function

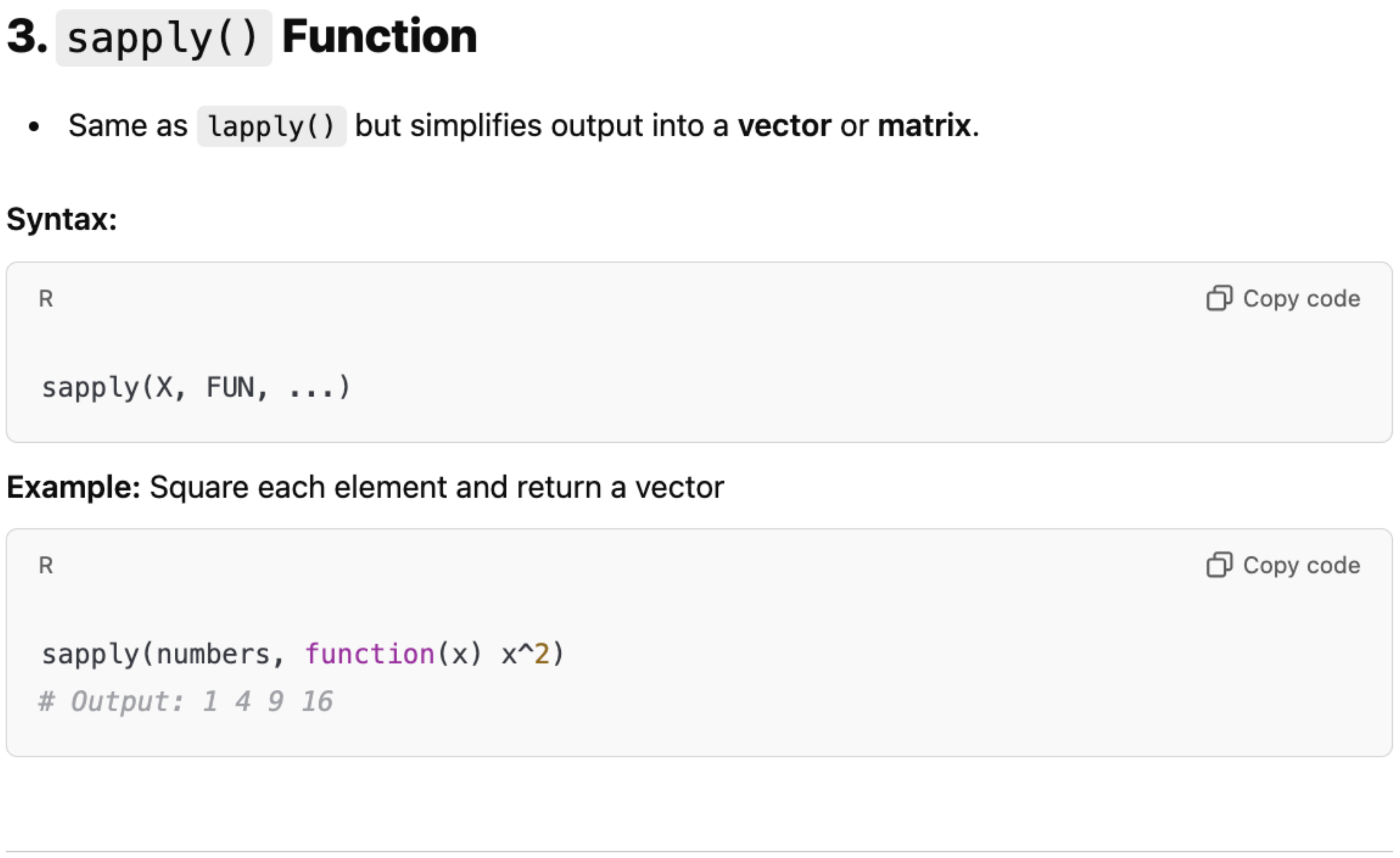
- Used for **lists** and **vectors**.
- Always returns a **list**.

Syntax:

```r
lapply(X, FUN, ...)
```

Example: Square each element in a list

```r
# Creating a list
numbers <- list(1, 2, 3, 4)

# Squaring each element
lapply(numbers, function(x) x^2)
# Output: [[1]] 1  [[2]] 4  [[3]] 9  [[4]] 16
```



3. `sapply()` Function

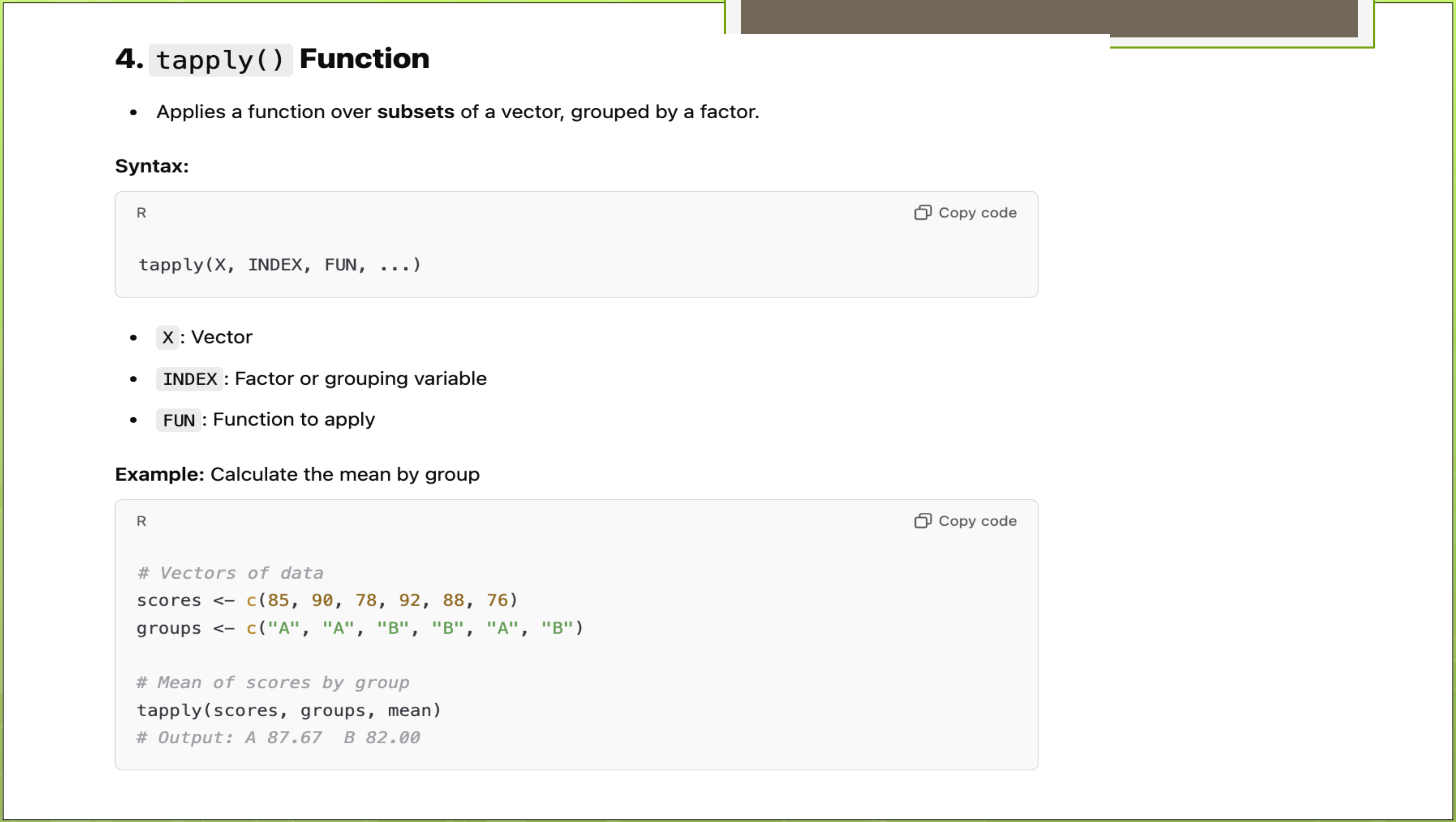
- Same as `lapply()` but simplifies output into a **vector** or **matrix**.

Syntax:

```r
sapply(X, FUN, ...)
```

Example: Square each element and return a vector

```r
sapply(numbers, function(x) x^2)
# Output: 1 4 9 16
```



4. `tapply()` Function

- Applies a function over **subsets** of a vector, grouped by a factor.

Syntax:

```R
tapply(X, INDEX, FUN, ...)
```

- `X` : Vector
- `INDEX` : Factor or grouping variable
- `FUN` : Function to apply

Example: Calculate the mean by group

```R
# Vectors of data
scores <- c(85, 90, 78, 92, 88, 76)
groups <- c("A", "A", "B", "B", "A", "B")

# Mean of scores by group
tapply(scores, groups, mean)
# Output: A 87.67  B 82.00
```

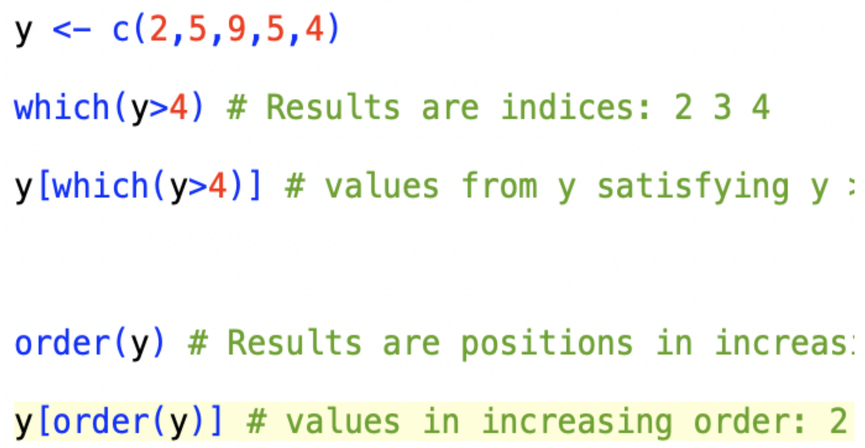
Other Functions

Which: Returns the *indices* of elements based on applied condition.

- **Usages**: filtering or locating positions of specific conditions in vectors or data frames.
- Syntax: *which(condition)*

Order: Returns the *indices* that would sort a vector or data frame.

- **Usages**: reordering data based on one or more variables.
- *Syntax: order(x, decreasing = FALSE)*



```
y <- c(2,5,9,5,4)

which(y>4) # Results are indices: 2 3 4

y[which(y>4)] # values from y satisfying y

order(y) # Results are positions in increas

y[order(y)] # values in increasing order: 2
```