R Programming

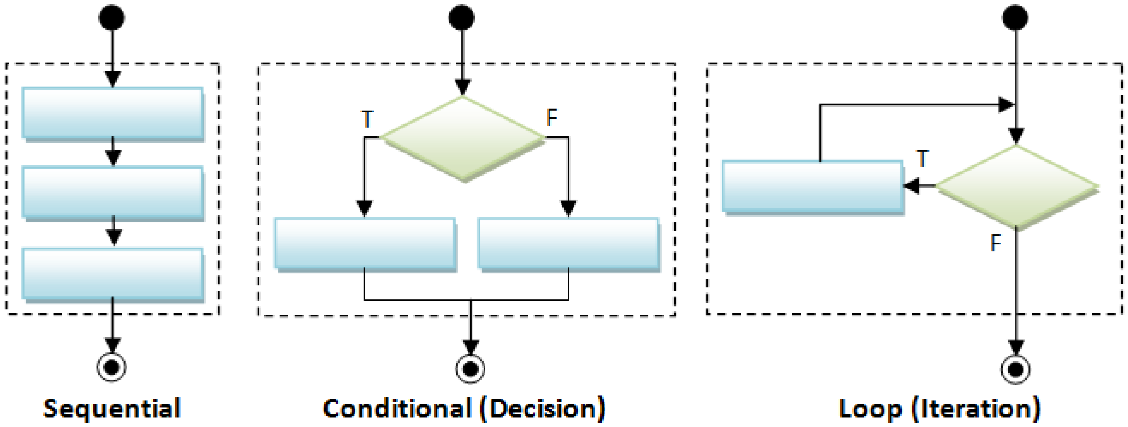
Table of contents

0.1 Agenda

1. Flow Control in R
2. Functions in R
3. Apply Functions
4. Exercises

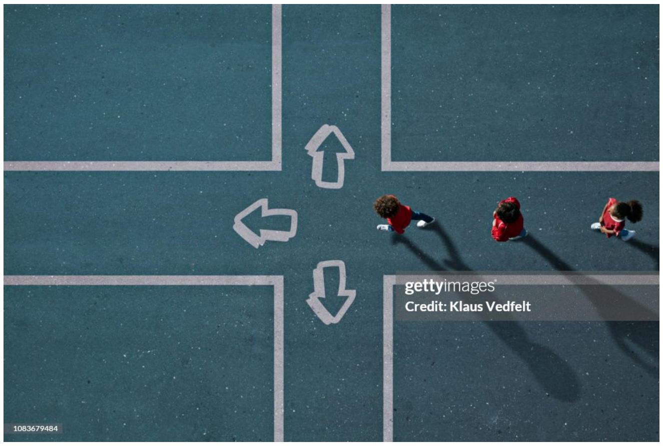
1 Flow Control in R

Flow control in R allows you to specify different paths of code execution based on conditions and repetitive structures.



1.1 Conditional

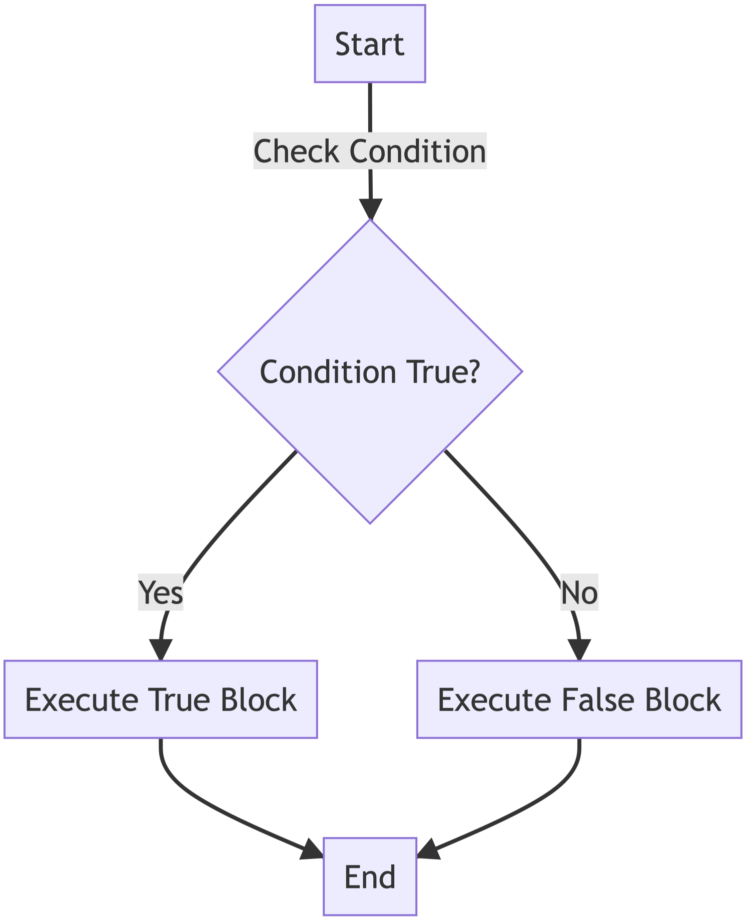
Conditions in R control the flow of execution in your program. Based on these conditions, different blocks of code may be executed.



1.2 `if` Statement in R

The `if` statement in R allows you to execute different blocks of code based on a condition.

1.2.1 Conditional Flowchart



1.2.2 Syntax

```
if (condition) {
  # code if true
} else {
  # code if false
}
```

1.2.3 Example

```
price = 20
if (price > 50) {
  category = "Expensive"
} else {
  category = "Affordable"
}

print(category)
```

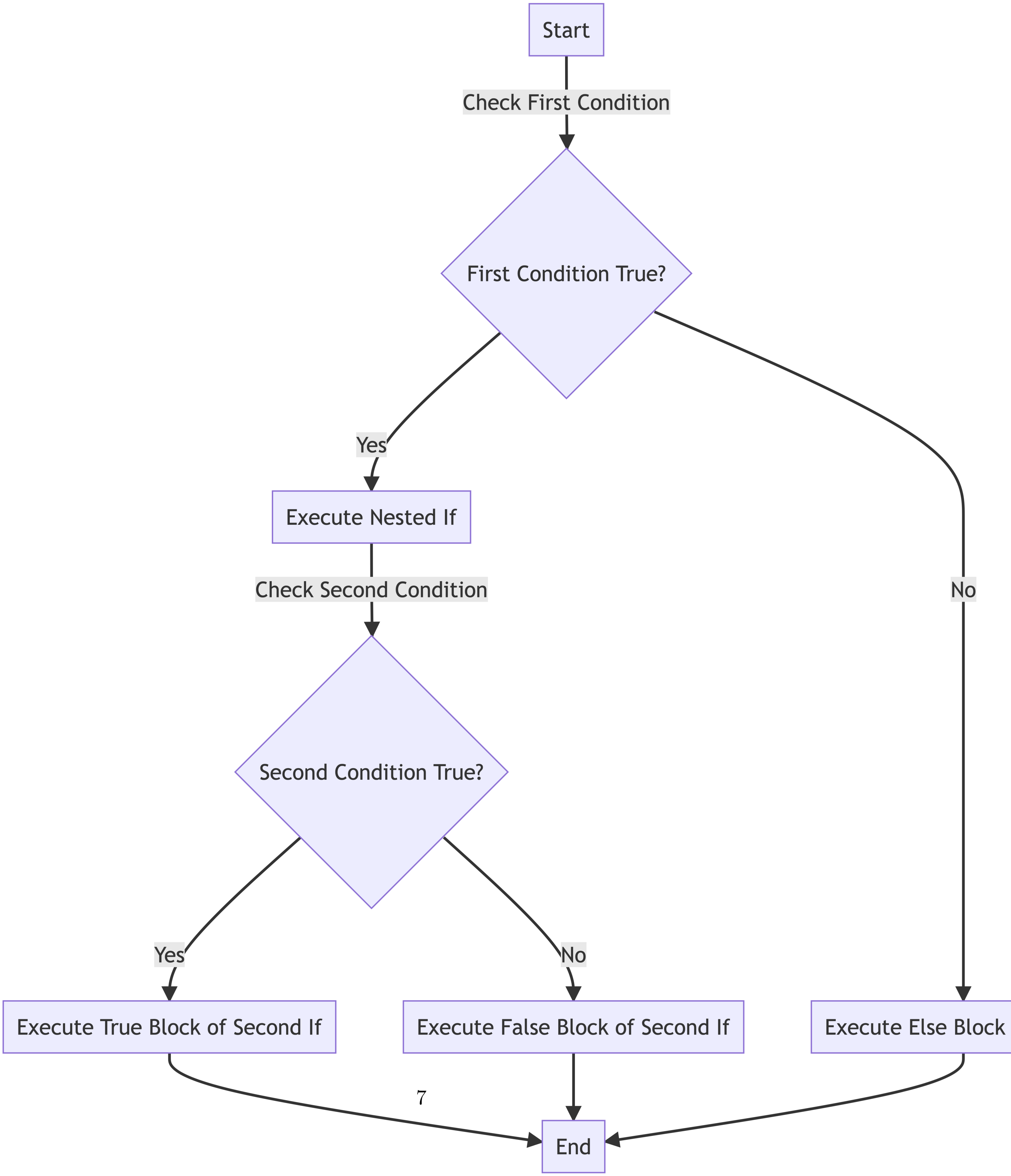
```
[1] "Affordable"
```

- **Explanation**: Based on the condition, the product is categorized as either `Expensive` or `Affordable`.

1.3 Nested `if` Statements

Nested `if` statements allow you to use an `if` statement inside another `if` statement.

1.3.1 Flowchart



1.3.2 Example

```
score = 85
if (score > 50) {
  if (score > 75) {
    grade = "A"
  } else {
    grade = "B"
  }
} else {
  grade = "F"
}

print(grade)
```

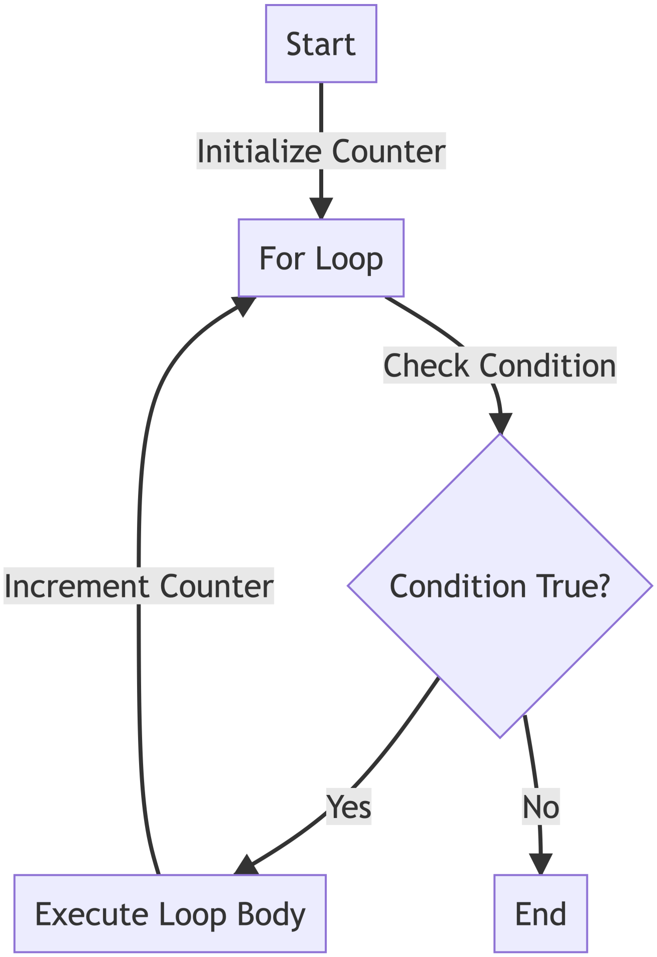
```
[1] "A"
```

- **Explanation**: The score is used to determine the grade of a student using nested `if` statements.

1.4 Looping

The execution of a block of code repeatedly for a specified number of times or until a particular condition is met

1.5 Looping Flowchart



1.6 `for` Loop

The `for` loop in R is used to iterate over a sequence of numbers or the elements of a vector.

- **Example**: Summing numbers in a sequence.

```r
numbers = 1:5
sum = 0
for (num in numbers) {
  sum = sum + num
}
```

```r
print(sum)
```

```
[1] 15
```

- **Explanation**: The sum of numbers from 1 to 5 is calculated using a `for` loop.

1.7 `while` Loops

The `while` loop in R repeatedly executes a block of code as long as a condition is *true*.

- **Example**:

```r
count = 1
while (count <= 5) {
  print(count)
  count = count + 1
}
```

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

- **Explanation**: This loop prints numbers 1 to 5.

1.8 The `break` Statement

- Use `break` to **exit** a loop prematurely.
- **Example**:

```r
count = 1
while (TRUE) {
  if (count > 5) break
  print(count)
  count = count + 1
}
```

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

- **Explanation**: This loop also prints numbers 1 to 5, but exits using `break`.

1.9 The `next` Statement

- Use `next` to **skip** the rest of the loop and start the next iteration.
- **Example**:

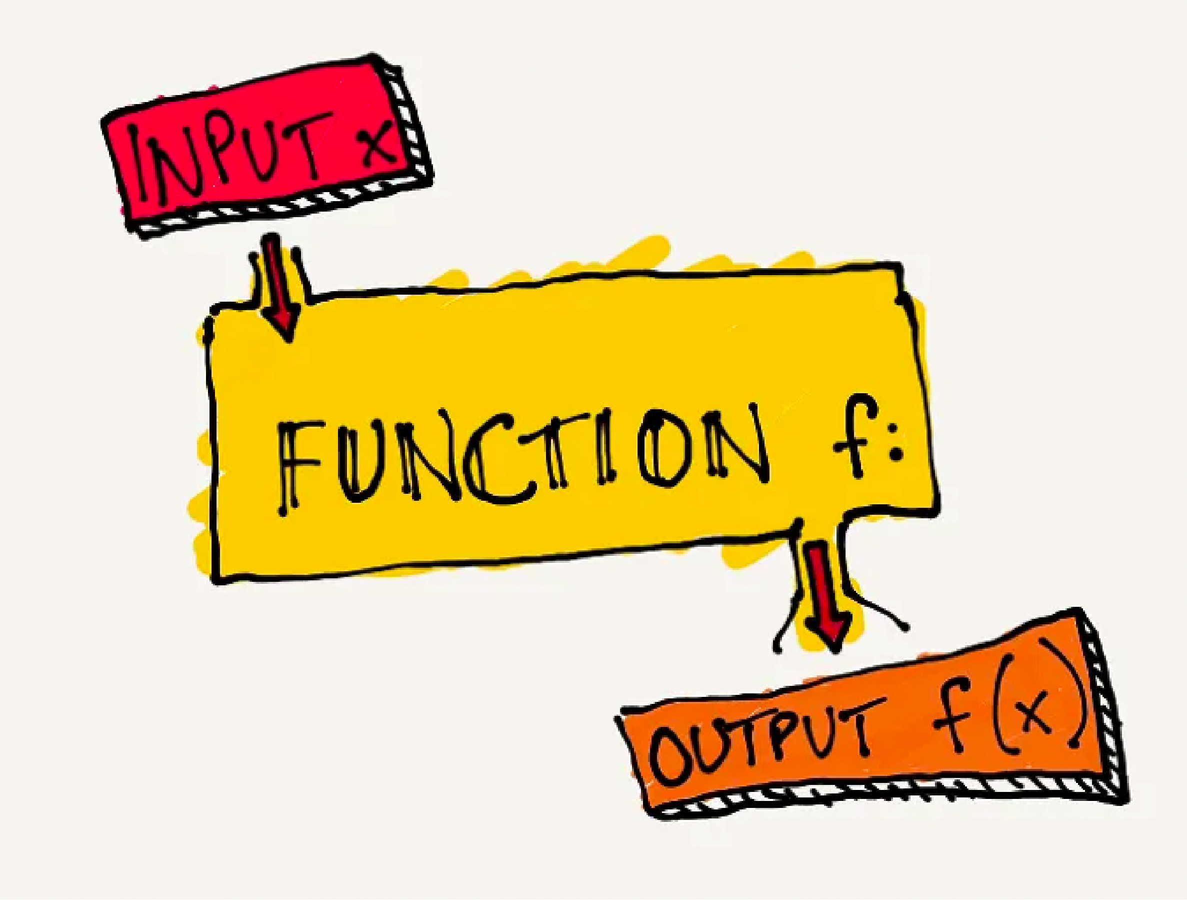
```r
for (i in 1:5) {
  if (i == 3)
    next
  print(i)
}
```

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

- **Explanation**: This loop prints numbers 1, 2, 4, and 5. Number 3 is *skipped*.

2 Functions in R

Functions in R are used to encapsulate code for **reusability** and **modularity**.



2.1 User-Defined Functions

User-defined functions in R allow you to create your own functions.

- **Syntax**:

```
  function_name = function(arg1, arg2, ...) {
    code
  }
```

- **Example**: Calculating the area of a rectangle.

```
calculate_area = function(length, width) {
  area = length * width
  return(area)
```

```r
}
area = calculate_area(10, 5)
print(area)
```

- **Explanation**: A function `calculate_area()` is defined to calculate the area of a rectangle given its length and width.

3 The Apply Functions Family

Apply functions in R provide a concise and efficient way to apply a function to the elements of data structures such as vectors, lists, data frames, or matrix.

3.1 Apply Functions

Apply functions provide a concise way to apply a function to data.

Function	Description	Usage	Example
`apply()`	Applies a function over the margins of an array or matrix.	`apply(X, MARGIN, FUN, ...)`	`apply(matrix(1:9, nrow = 3), 1, sum)`
`lapply()`	Applies a function to each element of a list, returning a list.	`lapply(X, FUN, ...)`	`lapply(list(1:5, 6:10), sum)`
`sapply()`	Similar to `lapply()`, but tries to simplify the result.	`sapply(X, FUN, ..., simplify = TRUE)`	`sapply(list(1:5, 6:10), sum)`

3.2 Example: Apply Functions

Calculate summary statistics for a list of numeric vectors.

```r
numeric_list = list(a = 1:5, b = 3:7, c = 10:14)
numeric_list
```

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

```
$b
[1] 3 4 5 6 7

$c
[1] 10 11 12 13 14
```

```r
lapply(numeric_list, mean)
```

```
$a
[1] 3

$b
[1] 5

$c
[1] 12
```

```r
sapply(numeric_list, sum)
```

```
 a  b  c 
15 25 60 
```

4 Exercises

4.1 Exercise 1: Grade Calculator

- Write an R function `calculate_grade()` to convert a numeric score to a letter grade.
- **Example:**
  - Input: `calculate_grade(85)`
  - Output: `"B"`
- **Solution**

```r
calculate_grade = function(score) {
  if (score >= 90) return("A")
  if (score >= 80) return("B")
  if (score >= 70) return("C")
  if (score >= 60) return("D")
  return("F")
}

calculate_grade(85)
```

```
[1] "B"
```

4.2 Exercise 2: Find Maximum

- Without using the R built-in function `max()`, write an R function `find_max()` to find the maximum in a numeric vector.
- **Example:**
  - Input: `find_max(c(2,5,4,1,3))`
  - Output: `5`
- **Solution:**

```r
find_max = function(numbers) {
  max_num = -Inf
  for (num in numbers) {
    if (num > max_num) {
      max_num = num
    }
  }
  return(max_num)
}

find_max(c(5,2,4,3))
```

```
[1] 5
```

- **Explanation**: The function iterates through the vector, keeping track of the maximum value found.

4.3 Exercise 3: Factorial using a `for` loop

- The factorial of a non-negative integer n, denoted as $n!$, is the product of all positive integers less than or equal to n.

$$n! = n \times (n-1) \times \cdots \times 1$$

- Write an R function `factorial()` to compute the factorial using a `for` loop.
- **Example:**
  - Input: `factorial(5)`
  - Output: `120`
- **Solution:**

```r
factorial = function(n) {
  product = 1
  for (i in 1:n) {
    product = product * i
  }
  return (product)
}

factorial(5)
```

```
[1] 120
```

4.4 Exercise 4: Factorial using a `while` loop

- Write an R function `factorial()` to compute the factorial of using a `while` loop.
- **Example:**
  - Input: `factorial(5)`
  - Output: `120`
- **Solution #1:** (moving backward)

```r
factorial = function (n) {
  product = 1
  while (n > 0) {
    product = product * n
    n = n - 1
  }
  return (product)
}
factorial(5)
```

```
[1] 120
```

- **Solution #2:** (moving forward)

```r
factorial = function (n) {
  product = 1
  i = 1
  while (i <= n) {
    product = product * i
    i = i + 1
  }
  return (product)
}
factorial(5)
```

```
[1] 120
```

4.5 Exercise 5: Loop Control

- Skip even numbers and stop if number is greater than 8 in a loop from 1 to 10.

- **Solution:**

```r
for (i in 1:10) {
  if (i %% 2 == 0) # Check for even numbers using the modulus operator %%
    next # Skip
  if (i > 8)
    break # Exit
  print(i)
}
```

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

4.6 Exercise 6: Printing a Pattern

- Write an R function `print_pattern()` to print the following pattern for a given number n. The pattern consists of numbers where each row contains the same number, and the number of times it appears is equal to its row number.
- **Example:** for $n = 5$, the pattern should look like this:

```
1
22
333
4444
55555
```

- Test your function with $n = 5$ and $n = 7$.
- **Solution:** The solution involves using **nested** loops. The **outer** loop iterates through the numbers 1 to n, and the **inner** loop prints the current number of the outer loop, as many times as the value of that number.

```r
print_pattern = function(n) {
  for (i in 1:n) { # The outer loop
    for (j in 1:i) { # The inner loop
      cat(i) # Print number
    }
    cat("\n") # Print newline
  }
}

# Test the function with n = 5
print_pattern(5)
```

```
1
22
333
4444
55555
```

```
# Test the function with n = 7
print_pattern(7)
```

```
1
22
333
4444
55555
666666
7777777
```

4.7 Exercise 7: Reverse Pyramid Pattern

- Write an R function `print_reverse_pyramid()` to print a reverse pyramid pattern for a given number n.
- **Example:** for $n = 5$, the pattern should look like this:

```
55555
 4444
  333
   22
    1
```

- Test your program with $n = 5$ and $n = 6$.
- **Solution #1:** The solution involves using **nested** loops. The **outer** loop iterates through the numbers from n to 1, and the **inner** loops are used for printing spaces and the numbers.

```r
print_reverse_pyramid = function(n) {
  for (i in n:1) { # Outer loop

    j = i
    while (j < n) { # Inner loop for the spaces
      cat(" ")
      j = j + 1 # print leading spaces
    }

    for (j in 1:i) { # Inner loop for the numbers
      cat(i) # print numbers
    }
    cat("\n")
  }
}

# Test the function with n = 5
print_reverse_pyramid(5)
```

```
55555
 4444
  333
   22
    1
```

```r
# Test the function with n = 6
print_reverse_pyramid(6)
```

```
666666
 55555
  4444
   333
    22
     1
```

- **Solution #2:** Instead of the inner loops, we can use the `rep()` function to generate the output

```r
print_reverse_pyramid = function(n) {
  for (i in n:1) {
    cat(rep(" ", n - i), sep = "")
    cat(rep(i, i), sep = "")
    cat("\n")
  }
}

# Test the function with n = 5
print_reverse_pyramid(5)
```

```
55555
 4444
  333
   22
    1
```

```
# Test the function with n = 6
print_reverse_pyramid(6)
```

```
666666
 55555
  4444
   333
    22
     1
```