Rtutorial-part2

March 10, 2025

1 Different kinds of loops in R

Loops offer a powerful way to go over each individual element stored in lists, vectors, or rows/columns in data frames. They are a great way to automate repetitive tasks.

Let us quickly explore the different types of loops in R: `for`, `while`, and `repeat`.

1.1 For Loop

A `for` loop is used to iterate over a sequence (such as a vector) and execute a block of code for each element.

1.1.1 Syntax

"'r for (variable in sequence) { # code to be executed }

[ ]:
```r
# Create a vector called numbers
numbers <- 1:5

# For loop to print each number
for (num in numbers) {
  print(num)
}
```

1.2 While Loop

A `while` loop is used to repeatedly execute a block of code as long as a condition is true.

1.2.1 Syntax

"'r while (condition) { # code to be executed }

[ ]:
```r
# Initialize a counter
counter <- 1

# While loop to print numbers from 1 to 5
while (counter <= 5) {
  print(counter)
  counter <- counter + 1
}
```

1.3 Repeat Loop

A `repeat` loop is used to repeatedly execute a block of code until a specific condition is met. It must be exited explicitly using a `break` statement.

1.3.1 Syntax

"'r repeat { # code to be executed if (condition) { break } }

[ ]:
```r
# Initialize a counter
counter <- 1

# Repeat loop to print numbers from 1 to 5
repeat {
  print(counter)
  counter <- counter + 1
  if (counter > 5) {
    break
  }
}
```

1.3.2 Summary

1. **For Loop**:
   - Iterates over each element in a sequence (e.g., vector).
   - Executes the block of code for each element.
2. **While Loop**:
   - Executes the block of code as long as the condition is `true`.
   - Requires the condition to be checked before each iteration.
3. **Repeat Loop**:
   - Executes the block of code indefinitely.
   - Must include a `break` statement to exit the loop based on a condition.

2 Conditional Statements in R

Let us now look at the use of `if` statements in R. Conditional statements allow you to execute certain pieces of code based on whether a condition is true or false.

2.1 If Statement

An `if` statement is used to test a condition and execute a block of code if the condition is true.

2.1.1 Syntax

"'r if (condition) { # code to be executed if condition is true }

[ ]:
```r
# Define a variable
x <- 10
```

```
# If statement to check if x is greater than 5
if (x > 5) {
  print("x is greater than 5")
}
```

An if-else statement allows you to execute one block of code if the condition is true and another block if the condition is false.

[ ]:
```
# Define a variable
x <- 3

# If-else statement to check if x is greater than 5
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

An if-else if-else statement allows you to test multiple conditions and execute different blocks of code for each condition. You can combine multiple `else if` statements if you want to test for even more conditions.

[ ]:
```
# Define a variable
x <- 7

# If-else if-else statement to check the value of x
if (x > 10) {
  print("x is greater than 10")
} else if (x > 5) {
  print("x is greater than 5 but less than or equal to 10")
} else {
  print("x is 5 or less")
}
```

You can also use nested `if` statements to check multiple conditions within a single block of code.

[ ]:
```
# Define a variable
x <- 8

# Nested if statements
if (x > 5) {
  if (x < 10) {
    print("x is between 6 and 9")
  } else {
    print("x is 10 or greater")
  }
} else {
  print("x is 5 or less")
```

```
}
```

2.1.2 Summary

- **If Statement**: Executes code if the condition is true.
- **If-Else Statement**: Executes one block of code if the condition is true, and another block if the condition is false.
- **If-Else If-Else Statement**: Tests multiple conditions and executes different blocks of code based on which condition is true.
- **Nested If Statements**: Allows multiple levels of condition checking within a single block of code.

3 R-centric techniques and quirks

[ ]:
```r
# Scalars are treated as 1-element vectors
# This means that even if you define a single number, it is treated as a vector␣
↪with one element.
x <- 1
print(is.vector(x))
print(length(x))
```

[ ]:
```r
# Vectors grow automatically as needed
# In R, vectors can grow automatically when you assign a value to an index that␣
↪is beyond the current length of the vector.
x[5] <- 2
print(x)
```

NA stands for “Not Available” or missing values.

[ ]:
```r
# How to allocate a vector of given length
# In R, you can create (allocate) a vector of a specified mode (data type) and␣
↪length using the vector function.
# Possible modes include "character", "logical", "integer", etc.
v <- vector(mode="numeric", length=10)
print(is.vector(v))
print(length(v))
```

Sequentially building up vectors or lists by appending elements one by one can be inefficient in R because it requires reallocating memory each time an element is added.

Instead, it’s more efficient to allocate the necessary memory once and then fill the vector or list.

[ ]:
```r
# avoid sequencial build-up of vectors or lists

# Inefficient way:
dynamic <- function(n) {
  v <- NULL                    # define empty object
```

```
  for (i in seq(n)) v <- c(v,i) # append element by element
  print(is.vector(v))
  return(v)
}

# Efficient way:
static <- function(n) {
  v <- vector(mode="integer", length=n) # allocate vector once
  for (i in seq(n)) v[i] = i            # fill it
  print(is.vector(v))
  return(v)
}

n = 50000

print("static ...")
system.time(static(n))
print("dynamic ...")
system.time(dynamic(n))
```

dynamic is inefficient because… * Initialization: v <- NULL initializes an empty vector. * Appending Elements: for (i in seq(n)) v <- c(v, i) appends elements to the vector one by one using c(v, i). * **Each iteration reallocates memory for v**, which is computationally expensive. * Check and Return: print(is.vector(v)) checks if v is a vector and prints the result. return(v) returns the final vector.

static is efficient because… * Initialization: v <- vector(mode="integer", length=n) pre-allocates a vector of length n with integer mode. * Filling Elements: for (i in seq(n)) v[i] <- i fills the vector with values from 1 to n. * **No memory reallocation is needed** during the filling process. * Check and Return: print(is.vector(v)) checks if v is a vector and prints the result. return(v) returns the final vector.

3.0.1 R quirks: no string indexing

In R, you cannot directly index individual characters in a string using square bracket notation as you might in some other languages.

Instead, you can use functions like substring() to extract parts of a string.

[ ]:
```
s <- "abcde"
substring(s, 1, 2)       # work around
```

3.0.2 R quirks: no strides in array indexing

In R, you cannot directly specify strides (step sizes) in the square bracket notation for indexing arrays.

Instead, you can use functions like seq() to create a sequence with a specific stride and then use that sequence for indexing.

[ ]:
```r
a <- seq(10)
a[seq(1,10,2)]          # work around
```

Explanation:

a <- seq(10): This creates a sequence from 1 to 10 and assigns it to the variable a.

seq(1, 10, 2): This generates a sequence from 1 to 10 with a step size of 2, resulting in c(1, 3, 5, 7, 9).

a[seq(1, 10, 2)]: This indexes a using the sequence generated by seq(1, 10, 2), effectively selecting every second element of a.

3.0.3 In R, lists are similar to Python dictionaries but maintain the order of elements, allowing for indexing by position.

[ ]:
```r
# Creating a list
alist <- list(one = 1, two = 2, three = 3)

# Printing key and value using single bracket indexing
cat("key and value:\n")
print(alist[1])

# Printing value only using double bracket indexing
cat("value only:\n")
print(alist[[1]])

# Accessing value via key using $ and double bracket notation
cat("value via key:\n")
print(alist$two)
print(alist[['two']])
```

[ ]:
```r
# looping over lists in R provides list values (not keys like for Python␣
 ↪dictionaries)
for (value in alist) print(value)
```

[ ]:
```r
# in contrast, here the keys are extracted with names()
for (key in names(alist)) print(key)
```

3.0.4 Use apply family of functions to avoid loops

Inefficiency: Loops in R, similar to Python, can be inefficient, especially when applied to large datasets.

This inefficiency arises because loops typically involve repeated interpretation and execution of R code, which can be slower compared to vectorized operations.

The `apply` family of functions in R (apply, lapply, sapply, etc.) can often replace loops.

These functions are more efficient because they are optimized for operations on arrays and lists.

They correspond to functions like map() and list comprehensions in Python.

[ ]:
```r
# Create a 10x5 matrix of random numbers
x <- matrix(rnorm(10*5), nrow=10, ncol=5)

# Apply the function "mean" to all columns and return the result in a vector
m <- apply(x, 2, mean)

# Print the matrix
print(x)
cat("\n")  # Print a newline for better readability

# Print the vector of column means
print(m)

# In this case, one could have used the built-in function colMeans() to do the␣
 ↪trick
```

3.0.5 What just happened here?

rnorm(10*5): Generates 50 random numbers drawn from a `normal` (Gaussian) distribution.

matrix(..., nrow=10, ncol=5): Reshapes these 50 numbers into a 10x5 `matrix`.

x is now a 10x5 `matrix` with random numbers.

`apply`(x, 2, `mean`): Applies the `mean` function to the second dimension (columns) of the matrix x.

The result is stored in m, which is a vector containing the mean of each column of the matrix x.

This is a flexible and powerful approach, as `apply` can take any function and apply it across dimensions of a matrix or array. However, for specific tasks like computing column means, it can be slightly less efficient than dedicated functions.

[ ]:
```r
# a more concise and efficient approach for computing column means is to use␣
 ↪built-in functions like colMeans:
m <- colMeans(x)
print(m)
```

The `replicate()` function in R is used to repeatedly evaluate an expression a specified number of times. It is related to `sapply()` in that both functions apply a function over a sequence, but `replicate()` is specifically designed for repeated evaluations of an expression, often used for simulations or random experiments.

[ ]:
```r
# toss_dice is a function that simulates rolling dice.

toss_dice <- function(n) sample(c(1,2,3,4,5,6), n, replace=TRUE) # function␣
 ↪contans a random component, n, which is the number of dice to roll
```

```r
# replace = TRUE: Indicates that the sampling is done with replacement, meaning␣
 ↪the same number can appear more than once in the result.

# Using replicate() to simulate dice tosses
x <- replicate(10, toss_dice(3))      # toss 3 dice 10 times
                                      # replicate attempts to combine the␣
 ↪results into a matrix or array

print(x)
```

3.0.6 The triple-dot argument

In R, the … argument in function definitions allows you to pass an arbitrary number of arguments to the function. These arguments can then be passed to other functions within the body of your function. This is particularly useful for creating flexible and general-purpose functions.

It functions similarly to ***args** and ****kwargs** in Python.

[ ]:
```r
add_random_numbers <- function(n, ...) {
    x <- rnorm(n, ...)
    return(sum(x))
}
```

add_random_numbers <- `function`(n, ...): This defines a `function` named add_random_numbers that takes an argument n and any additional arguments specified by ....

`rnorm`(n, ...): Generates `n` random numbers from a `normal` distribution, with additional parameters passed via ....

The `rnorm` function can take additional parameters like `mean` and `sd` to specify the mean and standard deviation of the normal distribution.

`return`(`sum`(x)): Calculates the `sum` of the generated random numbers and returns it.

[ ]:
```r
# Call the function with named arguments
print(add_random_numbers(5, mean=-5, sd=1.5))

# Call the function with positional arguments
print(add_random_numbers(5, 10, 1.5))
```

To know which optional arguments are available for a function like `rnorm`() in R, you can use the R help system. Specifically, you can use the `?` operator or the `help`() function to look up the documentation for `rnorm`(). This will provide a detailed description of the function, including all the optional arguments you can use.

[ ]:
```r
?rnorm
```

3.0.7 Defining your own custom operators

In R, you can define your own custom operators. This is a powerful feature that allows you to create operators that perform specific tasks, making your code more expressive and concise.

[ ]:
```r
# define an operator that concatenates two strings s1 and s2, replacing R's␣
↪paste() function
'%.%' <- function(s1,s2) paste(s1,s2,sep="") # This line defines a custom infix␣
↪operator %.%.
```

The syntax for defining a custom operator in R uses backticks (‘operator’) and the assignment operator <-.

The function body specifies what the operator does.

The `paste`() function in R concatenates strings with a specified separator. By default, paste() uses a space as a separator.

In this case, we use the `paste`() function to concatenate two strings s1 and s2 without any separator (i.e., we set `sep` = ””).

[ ]:
```r
# use it
'Hello' %.% ' ' %.% 'world!'
```

Our custom operator at work. First, it concatenates ‘Hello’ and ‘ ’ to produce a new string: ‘Hello’.

Then it is invoked again in sequence to combine the strings ‘Hello’ and ‘world!’, producing the new string ‘Hello world!’.

[ ]:
```r

```