Assessment 09 - Programming Basic

Alessandro Corradini - Harvard Data Science Professional

Conditionals

What will this conditional expression return? Run it from the console.

```
x <- c(1,2,-3,4)
if(all(x>0)){
    print("All Positives")
  } else{
     print("Not All Positives")
  }
```

Instructions

Possible Answers

- All Positives
- Not All Positives [X]
- N/A
- None of the above

Conditional continued

Which of the following expressions is always FALSE when at least one entry of a logical vector x is TRUE? You can try examples in the R console.

Instructions

Possible Answers

- all(x)
- any(x)
- any(!x)
- all(!x) [X]

ifelse

The function nchar tells you how many characters long a character vector is. For example:

```
char_len <- nchar(murders$state)
head(char_len)
```

The function `ifelse` is useful because you convert a vector of logicals into something else. For example, some datasets use the number -999 to denote NA. A bad practice! You can convert the -999 in a vector to NA using the following `ifelse` call:

```
x <- c(2, 3, -999, 1, 4, 5, -999, 3, 2, 9)
ifelse(x == -999, NA, x)
```

If the entry is -999 it returns NA, otherwise it returns the entry.

Instructions

We will combine a number of functions for this exercise.

Use the `ifelse` function to write one line of code that assigns to the object `new_names` the state abbreviation when the state name is longer than 8 characters. So, for example, where the original vector has Massachusetts

(13 characters), the new vector should have MA. But where the original vector has New York (8 characters), the new vector should have New York as well.

```
# Assign the state abbreviation when the state name is longer than 8 characters
new_names <- ifelse(nchar(murders$state) > 8, murders$abb, murders$state)
```

Defining functions

You will encounter situations in which the function you need does not already exist. R permits you to write your own. Let's practice one such situation, in which you first need to define the function to be used. The functions you define can have multiple arguments as well as default values.

To define functions we use `function`. For example the following function adds 1 to the number it receives as an argument:

```
my_func <- function(x){
    y <- x + 1
    y
}
```

The last value in the function, in this case that stored in `y`, gets returned.

If you run the code above R does not show anything. This means you defined the function. You can test it out like this:

```
my_func(5)
```

Instructions

We will define a function `sum_n` for this exercise.

- Create a function `sum_n` that for any given value, say `n`, creates the vector `1:n`, and then computes the sum of the integers from 1 to n.
- Use the function you just defined to determine the sum of integers from 1 to 5,000.

```
# Create function called `sum_n`
sum_n <- function(n){
  y <- sum(seq(1:n))
  y
}
# Use the function to determine the sum of integers from 1 to 5000
sum_n(5000)
```

Defining functions continued...

We will make another function for this exercise. We will define a function `altman_plot` that takes two arguments `x` and `y` and plots the difference `y-x` in the y-axis against the sum `x+y` in the x-axis.

You can define functions with as many variables as you want. For example, here we need at least two, `x` and `y`. The following function plots log transformed values:

```
log_plot <- function(x, y){
    plot(log10(x), log10(y))
}
```

This function does not return anything. It just makes a plot.

Instructions

We will make another function for this exercise.

Create a function `altman_plot` that takes two arguments `x` and `y` and plots `y-x` (on the y-axis) against `x+y` (on the x-axis).

```
# Create `altman_plot`
altman_plot <- function(x,y){
  plot(x+y, y-x)
}
```

Lexical scope

Lexical scoping is a convention used by many languages that determine when an object is available by its name. When you run the code below you will see which `x` is available when.

```
x <- 8
my_func <- function(y){
    x <- 9
    print(x)
    y + x
}
my_func(x)
print(x)
```

Note that when we define `x` as 9, this is inside the function, but it is 8 after you run the function. The `x` changed inside the function but not outside.

Instructions

The lexical scoping is a convention used by many languages that determine when an object is available by its name. When you run the code below you will see which `x` is available when. After running the code below, what is the value of `x`?

```
x <- 3
my_func <- function(y){
    x <- 5
    y
    print(x)
}
my_func(x)
```

```
# Run this code
x <- 3
    my_func <- function(y){
    x <- 5
    y+5
}

# Print value of x
print(x)
```

For loops

In the next exercise we are going to write a for-loop. In that for-loop we are going to call a function. We define that function here.

Instructions

- Write a function `compute_s_n` that for any given n computes the sum $ S_n=1^{2+}2^{2+}3^{2+\ldots+n}2 $.
- Report the value of the sum when `n=10`.

```
# Here is an example of function that adds numbers from 1 to n
example_func <- function(n){
    x <- 1:n
    sum(x)
}

# Here is the sum of the first 100 numbers
example_func(100)

# Write a function compute_s_n that with argument n and returns of 1 + 2^2 + ...+ n^2
compute_s_n <- function(n){
  y <- sum(seq(1:n)^2)
  y
}
# Report the value of the sum when n=10
compute_s_n(10)
```

For loops continued...

Now we are going to compute the sum of the squares for several values of $ n $. We will use a for-loop for this. Here is an example of a for-loop:

```
results <- vector("numeric", 10)
n <- 10
for(i in 1:n){
    x <- 1:i
    results[i] <- sum(x)
}
```

Note that we start with a call to `vector` which constructs an empty vector that we will fill while the loop runs.

Instructions

- Define an empty numeric vector `s_n` of size 25 using `s_n <- vector("numeric", 25)`.
- Compute the the sum when n is equal to each integer from 1 to 25 using the function we defined in the previous exercise: `compute_s_n`
- Save the results in `s_n`

```
# Define a function and store it in `compute_s_n`
compute_s_n <- function(n){
  x <- 1:n
  sum(x^2)
}

# Create a vector for storing results
s_n <- vector("numeric", 25)

# write a for-loop to store the results in s_n
for (i in 1:25){
  s_n[i] <- compute_s_n(i)
}
```

Checking our math

If we do the math, we can show that

$S_n = 1^2 + 2^2 + 3^2 + ... + n^2 = n(n+1)(2n+1)/6$

We have already computed the values of S_n from 1 to 25 using a for loop.

If the formula is correct then a plot of S_n versus n should look cubic.

Let's make this plot.

Instructions

- Define `n <- 1:25`. Note that with this we can use `for(i in n)`
- Use a for loop to save the sums into a vector `s_n <- vector("numeric", 25)`
- Plot `s_n` (on the y-axis) against `n` (on the x-axis).

```
# Define the function
compute_s_n <- function(n){
  x <- 1:n
  sum(x^2)
}

# Define the vector of n
n <- 1:25

# Define the vector to store data
s_n <- vector("numeric", 25)
for(i in n){
  s_n[i] <- compute_s_n(i)
}

#  Create the plot
plot(n, s_n)
```

Checking our math continued

Now let's actually check if we get the exact same answer.

Instructions

Confirm that `s_n` and $n(n+1)(2n+1)/6$ are the same using the `identical` command.

```
# Define the function
compute_s_n <- function(n){
  x <- 1:n
  sum(x^2)
}

# Define the vector of n
n <- 1:25

# Define the vector to store data
s_n <- vector("numeric", 25)
for(i in n){
  s_n[i] <- compute_s_n(i)
}

# Check that s_n is identical to the formula given in the instructions.
identical(s_n, n*(n+1)*(2*n+1)/6)
```