0-Based Indexing: A Julia How-To

Soumitra Shewale

Introduction

Julia, unlike the majority of the languages, is 1-based instead of 0-based. This means that arrays start at index 1. This causes significantly different algorithms and loop designs. To see the merits of each system and some use cases, refer to this document: Indexing of Arrays: 0 vs 1.

In this document, we will explore one way to change the indices of arrays in Julia to fit our needs. We will use the OffsetArrays.jl package to do the same. The GitHub repo for it can be found here.

Using examples built-in Julia, we will walk through the process of creating a custom indexed array in Julia.

Installing the Package

In your REPL, (accessed by typing “julia” on the command line, and only works if Julia is on your PATH) type:

```
import Pkg
```

Then, you can install the OffsetArrays packages by using:

```
Pkg.add("OffsetArrays")
```

Importing the Package

To import the package, simply type

```
using OffsetArrays
```

The package is now imported, and ready for use.

Basic Usage

To create a custom indexed array after importing the package, type:

```
array_name =  OffsetVector([element1, element2, element3, element3],
start_index:end_index)
```

Where element1, 2, 3, etc are the elements of the array and start_index is the index of the initial element of the array, and end_index is the index of the last element.

We can even create Multi-Dimensional Arrays using this package that have custom indices like this:

```
array_name = OffsetArray{Float64}(undef, -1:1, -7:7)
```

This creates a 2D array with 3 rows (indexed -1, 0, and 1 each), and 15 columns (indexed -7, -6, -5 and so on until 7) containing all #undef objects.

Example: Circular List

Let us implement a circular list without using any pointers like a circularly linked list. We will NOT create an object, we will only write a function in which accesses a 0-based array circularly.

The code has comments and explains our algorithm:

```
using OffsetArrays


# Create a 0-indexed array

a = OffsetVector(['a', 'b', 'c', 'd'], 0:3)


# 1-based array to compare with and show that it does not work

b = ['a', 'b', 'c', 'd']


# Create a function that takes in an array, and cycles through it
cycle_count times

function print_circular(array, cycle_count)
```

```
    # Loop through values of j ranging from 0 to
size_of_array*cycle_count-1. This makes j complete the correct number of
cycles.

    # NOTE: it says size(array)[1] because the size function returns a
tuple with the 2D dimensions of the array.

    #       Since we are working with a 1D vector, we are only interested
in its x-dimension. This happens to be the 1st element.

    #       The tuple returned is 1-based since the core functions of
Julia still use 1-based indexing. We have only created a custom

    #       array. We have not changed the behaviour of Julia.

    for j in 0:(size(array)[1]*cycle_count)-1

        # Print the element at the j%size th position. This is what gives
us the cyclic behaviour. As j increases to any number,

        # the element accessed is the j%size th element. So, j snaps back
to 0 when j becomes a multiple of the size. This means

        # that at the end of a cycle, the 0th element is accessed. This is
the reason 0-based indexing is necessary.

        print(array[j%size(array)[1]])



        # If we are at the last step of a cycle, print a space character.
This is only to make the output easier to see, and the

        # cycles easy to count. This does not affect the circular list in
any way.

        if j%size(array)[1] == (size(array)[1]-1)

            print(" ")

        end

    end
```

```
end


# Test the function on our custom 0-based array and also on our 1-based
array for comparison

print_circular(a, 5)

println("\n")

print_circular(b, 5)


# OUTPUT:

# The 0-based array works fine, while the 1-based array throws an error.
```

A Final Note

Again, as mentioned in the code, we aren't changing the behavior of Julia. We can still create arrays in the normal way, and they will start at index 1, when a function returns a tuple or array, its index will still start at 1. We have only created a custom object, which works exactly like an array, and has custom indices instead of the conventional 1-based indexing.