Lesson 8, Week 1: Data Containers I (strings)

AIM

— To introduce the idea of a data container, using strings as the example

— To introduce arrays as data containers

— To explain comprehensions as a way to make an array

After this lesson, you will be able to

* Say what a data container is, in general terms
* Use indexing to extract characters from strings
* Use a range as index to extract a string from a string
* Explain what it means for a character to use more than one code unit
* Explain the difference between the functions `length` and `ncodeunits` as applied to strings
* Use a comprehension to write the characters of a string to an array
* Say what an array is, in general terms

Strings as data containers

What is a container? You'll see this word quite often if you ever start reading the Julia documentation. "Container" is a slightly vague term for anything that may contain more than one value. But of course a container is itself a value, so a good way to define them is "a data container is a value that may contain more than one value". Strings are our first example: a string value can contain more than one character value[1].

By the way, containers are absolutely essential when it comes to organising the data on computers. All high-level computer languages[2] offer the programmer several kinds of containers.

[1] Recall that an empty string is valid in Julia, but an empty character is not. Essentially, this is because strings are containers and characters are not.

[2] "High-level" doesn't mean they're special, better than others. In computer science, low-level operations are those that are very close to the fine detail of what actually happens inside a computer. A low-level language is one that deals only or almost only with low-level operations. A high level language is basically a way to translate human ideas into low-level language.

By seeing a string as a container, you realise that you need ways to get the data into the container—everything you've so far learnt about strings is about that. However, you also need ways to get data out of the container, and this lesson is about that.

Indexing into strings

Indexing is one way to get data out of a string[3]. The idea is that in a string there's a first character, a second character, a third, and so on. This can function in a way similar to house numbers in a street: `eg1 = "abcde"; eg1[1]`

Point 1: you tell Julia the value of the index by delimiting with square brackets. Point 2: in this case, indexing works perfectly, that is, `eg1[k]` with give us the `k`-th character in `eg1`. Provided of course that the value of `k` is one of `1, 2, 3, 4, 5`; anything else throws an error.

Let me emphasise again the use square brackets in this case, as opposed to the round parentheses you've seen before. In Julia, it is very important to use the correct delimiters!

You can index into a string directly: `"a1b2"[3]`.

Finally, indexing can be to part of a string (pay careful attention to how the colon is used)
`greeting = "Hello, world"; greeting[2:8]`

The code fragment is `2:8` here is a range[4]. It starts with the index value 2 and ends with the index value 8. You can use variables to construct a range, assign the result to a variable, and use that variable to index a string:
`init, last = 1, 3; myrange = init:last; "1a2b3c"[myrange]`

A range can use steps[5] bigger than 1: `greeting[3:2:7]`, which extracts the 3rd, 5th and 7th characters. You can even use negative steps, as in reversing the order: `greeting[end:-1:1]`.

Note that the result of indexing with a range is always a string, even if the range starts and ends with the same value: `greeting[2:2]`

In other words, to get a single character out of a string, you use the correct index, not a range. To get that same character as a string, index with a range that starts and ends with the same index value.

[3] All of the data you get out will consist of characters, of course, because that is the only kind of data inside a string.

[4] Range one of several ways to index to several values at the same time. In this course, we do not explore the topic of indexing into containers in any depth at all. However, expert use of Julia requires a good knowledge of all the possibilities.

[5] Many people use the word *stride* for the step length in a range

A complication: the difference between `length` and `ncodepoints`

Sadly, not all strings index as nicely as `eg1`. That is Julia uses characters with variable width[6]—some take more memory to store than others. In Julia, this is expressed in terms of the number of code units the character requires for storing in memory. Each character in `eg1` requires just one code unit. The function `length`, when applied to a strings, returns the number of characters in the string. The function `ncodeunits` returns the number of code units.

Because `eg1` uses characters which are a one code unit wide, both `length(eg1)` and `ncodeunits(eg1)` return the value `5`.

The Greek alphabet is an example of where they differ [DEMO reminder: how to enter Greek letters; `length("α")` vs `ncodeunits(α)`]

This means that `"α"[1]` is valid code, but `"α"[2]` not. [DEMO: this is not a bounds error].

Similarly, for `eg2 = "αβ"` we have that `eg2[1]` and `eg2[3]` are valid code, but `eg2[2]` is not, nor is `eg2[4]`. The rule is: if the index points to the first code unit of a character, the whole character is returned. If it points to any other code unit, an error is thrown.

Moreover, if a range starts and ends with an index that corresponds to a character, that index is valid. Here, `eg2[1]` and `eg2[3]` are valid and hence so is `eg2[1:3]`.

For completeness: `ncodeunits` can take a single character as input and give its width. [DEMO]

Solution: use a comprehension to make an array of the characters

If you need the characters of a string separately, you can make an array of them. In that case, the variable width doesn't matter. There are many ways to do so, but using a comprehension is by far the most convenient:
`greeting_array = [char for char in greeting]`.
This is an assignment, i.e. a left hand side name linked by an equals sign to a right hand side value.

The right hand side is your first bit of tricky code. It is worth paying it very careful attention. There are a whole lot of technical issues to deal with:

First Please be aware the the word "Array" is a technical term in Julia, with a very precise meaning. Like a string, it is a container in which elements can be accessed by means of indexing. It is different from a string in two ways: you can change the values inside an array, and any kind of value can be used, whereas a string can only contain valid characters.

[6] So do other languages, but they may handle the issue differently. There is no way around it, really. Computers these days are dominated by variable-width encodings of characters. You can read more about this in the Julia documentation, where there is also a brief discussion of the reasons in favour of doing it the Julian way.

Second The square brackets as delimiters is how we know that we are telling Julia to create an array.

Third The name `char` inside the square brackets occurs twice because it has two different tasks.

Fourth The "for" in the square brackets is a reserved keyword[7]. This means that it cannot be used as a name, despite being validly formed.

Fifth The `in` in the square brackets is an operator whose meaning we explain below. It is not reserved, so it can be used as a name[8].

That expression, from the opening square bracket to the closing square bracket, is a comprehension. The square brackets indicate that the value will be an array. The name `char` is purely temporary, and doesn't continue to exist after the assignment is completed.

How does the comprehension work? It is quite simple: the second `char` is a variable that takes the values in `greeting` one after the other. We say that it iterates[9] over the values in `greeting`. The first `char` creates the value that is written into the array. Once the iteration is over, the resultant array is assigned to `greeting_array`. [DEMO: create other values to write into the array]

Question: how does Julia know to iterate inside a comprehension? Answer: the combination of the keyword `for` and the operator `in`. So this combination is the only way to create a comprehension, and it only works inside square brackets, because comprehensions are used only to make arrays[10].

Let's summarise: the syntax of a comprehension is `[<value> for <dummyvar> in <container>]`. Here, `<container>` indicates the container you iterate over, `<dummyvar>` indicates that you need a variable to hold each value one by one as it is extracted, and `<value>` indicates the expression used to create the values that goes into the array. The square brackets, the `for` and the `in` are always used in the same way to make a comprehension.

A few more introductory remarks about arrays

Arrays are a very efficient way to store memory in a computer, particularly when the elements of the array have fixed width[11].

Arrays like `greeting_array` are said to be one-dimensional, because they're like a street, you need only one number to index a value. But often it is convenient to use more dimensions, for example an appointment for nine o'clock on 13 March has three dimensions: the month, the day and the time. In some computer applications, the number of dimensions gets very large, but many of the elements are not used. Such an array is said to be sparse[12], and efficiently working with sparse arrays is one of Julia's great strengths.

[7] You can see the list of reserved keywords at `https://docs.julialang.org/en/v1/base/base/#Keywords-1`

[8] It might even be a good name in some contexts, but short names like this should be avoided, unless you have clear and good reason for using them.

[9] Iteration is very important in computer programs, and we will see it again.

[10] This is typical of Julia: specialised and very compact code to create a frequently used container. Almost all computer languages have ways to write specialised and compact code.

[11] This means that an array of characters is not actually the most efficient way to store a string, which is why Julia has a separate way to store strings.

[12] Storing one's appointments for a whole year in a three-dimensional array is almost certain to result in a very sparse array!

We will not on this course go into the details of how to use arrays efficiently in Julia. This is partly because it is an advanced topic, but mostly because Julia provides a very large number of ways to deal with arrays, each of which applies only to a few specialised cases. That is, for a particular application, it is worth learning efficient array handling. However, it is quite unpredictable whether in the next application you take on, efficient array handling will work the same way. This is one of those areas in programming where we believe it is not helpful to learn everything[13].

Review and summary

This has been your longest and most technical lecture so far, with several new ideas. Please make sure that you go through exercises. If you cope with them you are well on your way to being a Julia programmer!

* Strings and arrays are data containers
* The data in a string can be extracted using indexing using a pair of square brackets as the delimiters.
* Indexing into a string can a single integer or a range
* A range index into a string returns another string
* The width of a character is the number of code units used to store that character
* Characters in Julia have variable width; use the function `ncodeunits` to determine width
* `length` returns the number of characters in a string; `ncodeunits` returns the width
* A comprehension creates an array by iterating over a container
* A comprehension consist of square brackets around an iteration
* The iteration in a comprehension uses the reserved keyword `for` and the operator `in`
* The syntax of a comprehension is `<value> for <dummyvar> in <container>`
* Arrays in general are an advanced topic we do not take on in this course

[13] Unless you want to specialise in array handling, of course!