Applying Strings, Arrays and Slices

This lesson discusses relationship of slices with arrays and strings.

WE'LL COVER THE FOLLOWING

Making a slice of bytes or runes from a string

A string in memory is, in fact, a 2 word-structure consisting of a pointer to the string data and the length. The pointer is completely invisible in Go-code. For all practical purposes, we can continue to see a string as a value type, namely its underlying array of characters. If `s` is a string (so also an array of *bytes*), a slice of bytes `c` can immediately be made with

```
c:=[]byte(s)
```

This can also be done with the `copy`-function:

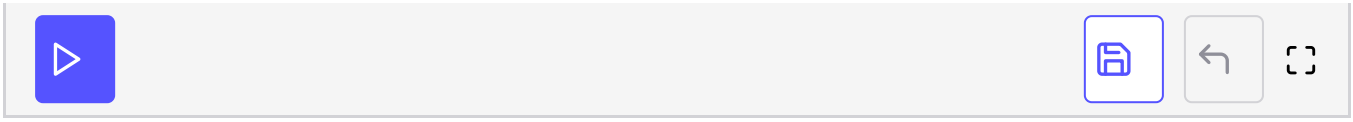
```
copy(dst []byte, src string)
```

The for `range` can also be applied as in the following program:

```
package main
import "fmt"

func main() {
  s := "\u00ff\u754c"
  for i, c := range s {
```

```
    fmt.Printf("%d:%c ", i, c)
  }
}
```



Strings with For range

We see that with `range` at **line 6** the returned character `c` from `s` is in Unicode, so it is a *rune*. Unicode-characters take **2** bytes; some characters can even take **3** or **4** bytes. If erroneous UTF-8 is encountered, the character is set to **U+FFFD** and the index advances by *one* byte.

In the same way as above, the conversion:

```
c:=[]int(s)
```

is allowed. Then, each int contains a Unicode code point, meaning that every character from the string corresponds to one integer. Similarly, the conversion to runes can be done with:

```
r:=[]rune(s)
```

The number of characters in a string `s` is given by `len([]int(s))`. A string may also be appended to a byte slice, like this:

```
var b []byte
var s string
b = append(b, s...)
```

Making a substring of a string

The following line:

```
substr := str[start:end]
```

takes the substring from `str` from the byte at index `start` to the byte at index `end- 1`. Also, `str[start:]` is the substring starting from index `start` to `len(str)` – 1, and `str[:end]` is the substring starting from index **0** to `end` – 1.

Changing a character in a string

Strings are immutable. This means when `str` denotes a string then `str[index]` cannot be the left side of an assignment:

```
str[i] = 'D'
```

Where `i` is a valid index, it gives the error `cannot assign to str[i]`.

To change a character, you first have to convert the string to an array of bytes. Then, an array-item of a certain index can be changed, and the array must be converted back to a new string. For example, change “hello” to “cello”:

```
s := "hello"
c := []byte(s) // converting string s to array of bytes `c`.
c[0] = 'c'     // modifying the c
s2 := string(c) // Converting c to string, s2 == "cello"
```

It is clear that string-extractions are very easy with the slice-syntax.

Searching and sorting slices and arrays

Searching and sorting are very common operations, and the standard library provides these in the package `sort`. To sort a slice of ints, import the package `sort` and simply call the function `func Ints(a []int)` as:

```
sort.Ints(arr)
```

where `arr` is the array or slice to be sorted in ascending order. To test if an array is sorted, use:

```
func IntsAreSorted(arr []int) bool
```

This returns **true** or **false** whether or not the array `arr` is sorted. Similarly, for float64 elements, you use the following function from `sort` package :

```
func Float64s(a []float64)
```

and for strings, there is a function provided by `sort` package:

```
func Strings(a []string)
```

```
func Strings(a []string)
```

To search an item in an array or slice, the array must first be *sorted* (the reason is that the search-functions are implemented with the binary-search algorithm). Then you can use:

```
func SearchInts(a []int, n int) int
```

This searches for `n` in the slice `a`, and returns its *index*. Of course, the equivalent functions for float64s and strings exist also:

```
func SearchFloat64s(a []float64, x float64) int // search for float64
func SearchStrings(a []string, x string) int // search for strings
```

Further details can be found in the official information.

Simulating operations with `append`

The `append` method, as we studied previously, is very versatile and can be used for all kinds of manipulations:

- Append a slice `b` to an existing slice `a`:

  ```
  a = append(a, b...)
  ```

- Delete item at index `i`:

  ```
  a = append(a[:i], a[i+1:]...)
  ```

- Cut from index `i` till `j` out of slice `a`:

  ```
  a = append(a[:i], a[j:]...)
  ```

- Extend slice a with a new slice of length `j`:

  ```
  a = append(a, make([]T, j)...)
  ```

- Insert item x at index `i`:

  ```
  a = append(a[:i], append([]T{x}, a[i:]...)...)
  ```

- Insert a new slice of length `j` at index `i`:

```
a = append(a[:i], append(make([]T, j), a[i:]...)...)
```

- Insert an existing slice `b` at index `i`:

```
a = append(a[:i], append(b, a[i:]...)...)
```

- Pop highest element from stack:

```
x, a = a[len(a)-1], a[:len(a)-1]
```

- Push an element x on a stack:

```
a = append(a, x)
```

So to represent a resizable sequence of elements use a slice and the `append` function. A slice is often called a **vector** in a more mathematical context. If this makes code clearer, you can define a *vector* alias for the kind of slice you need.

Slices and garbage collection

A slice points to the underlying array. This array could potentially be much bigger than the slice. As long as the slice is referred to, the full array will be kept in memory until it is no longer referenced. Occasionally, this can cause the program to hold all the data in memory when only a small piece of it is needed. For example, the following `FindDigits` function loads a file into memory and searches it for the first group of consecutive numeric digits, returning them as a new slice:

```
var digitRegexp = regexp.MustCompile("[0-9]+")
func FindDigits(filename string) []byte {
  b, _ := ioutil.ReadFile(filename)
  return digitRegexp.Find(b)
}
```

This code works as described, but the returned **[ ]byte** points into an array containing the entire file. Since the slice references the original array, as long as the slice is kept around, the garbage collector can't release the array; the

few useful bytes of the file keep the entire contents in memory.

To fix this problem one can copy the interesting data to a new slice before returning it:

```
func FindDigits(filename string) []byte {
  b, _ := ioutil.ReadFile(filename)
  b = digitRegexp.Find(b)
  c := make([]byte, len(b))
  copy(c, b) // copying b to c
  return c
}
```

Now, the `copy` function is copying `b` to `c`,which means slice is not referring original array. In this case, garbage collector can release the array if needed.

As an example, look at the following program that traverses an array of characters, and copies them to another array only if the character is different from that which precedes it.

```
package main
import (
        "fmt"
)

var arr []byte = []byte{'a','b','a','a','a','c','d','e','f','g'}

func main() {
        arru := make([]byte,len(arr)) // this will contain consecutive non-repeating items
        ixu := 0 // index in arru
        tmp := byte(0)
        for _, val := range arr {
                if val!=tmp {
                        arru[ixu] = val
                        fmt.Printf("%c ", arru[ixu])
                        ixu++
                }
                tmp = val
        }
}
```

Removing Consecutive Repeating Characters

In the code above, at **line 6**, we make a global array `arr` of type `byte` and initialize it with some values. In `main`, we make an array `arru` of type `byte`

with the same length as `arr` via the `make` function at **line 9**. Now, we maintain an index `ixu` to traverse through the array `arr` at **line 10**. In the next line, we declare a `byte` type variable `tmp` to keep track of the previously visited element of `arr`.

Now, we have a for loop at **line 12** that will iterate `len(arr)` times. We add a condition: `if val!=tmp` at **line 13**. The `val` contains the recently visited variable and `tmp` contains the element visited before `val`. We set `arr[uix]` to `val` because `val` holds a unique element. Then, it increments the index `iux` if the condition is satisfied. Whether the condition is satisfied or not, we set `val` to `tmp` to keep a record of the previously visited value before moving forward. So, in the end, you'll have all the unique values in `arru`.

Slices are interactive and flexible. We can use arrays and strings to make slices work as explained in detail in this lesson. In the next lesson, you will write a function to solve a coding problem.
