Slices

This lesson describes some important concepts about slices, i.e., how to use, declare and initialize them.

Concept

A **slice** is a reference to a contiguous segment (section) of an array (which we will call the underlying array, and which is usually anonymous), so a slice is a reference type (more like the array type in C/C++, or the list type in Python). This section can be the entire array or a subset of the items indicated by a start and an end index (the item at the end index is not included in the slice). Slices provide a dynamic window to the underlying array.

A slice in memory is a structure with **3** fields:

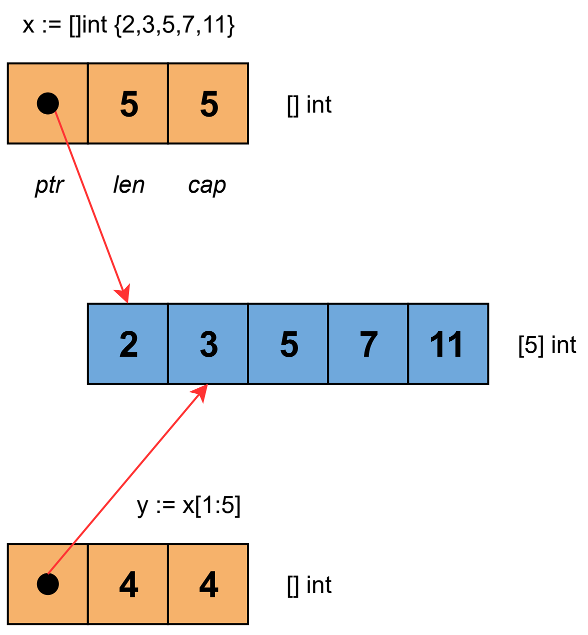
- a pointer to the underlying array
- the length of the slice
- the capacity of the slice

Slices are *indexable* and have a length given by the `len()` function. The slice-index of a given item can be less than the index of the same element in the underlying array. Unlike an array, the length of a slice can change during the execution of the code, minimally 0 and maximally the length of the underlying array, which means a slice is a variable-length array.

The built-in capacity function `cap()` of a slice is a measure of how long a slice can become. It is the length of the slice + the length of the array beyond the slice. If `s` is a slice, *cap* is the size of the array from **s[0]** to the end of the array. A slice's length can never exceed its capacity, so the following statement is always true for a slice `s`:

```
0 <= len(s) <= cap(s)
```

This is illustrated in the following figure, where the slice `x` has a capacity and length of **5**. Slice `y`, on the other hand, is of length **2** and capacity **4**. The `y` slice takes its value from slice `x`. The slice `y` is equal to `x[1:5]` and contains the elements 3, 5, 7 and 11.



Slice in Memory

Multiple slices can share data if they represent pieces of the same array, but multiple arrays can never share data. A slice, therefore, shares storage with its array and with other slices of the same array. In contrast, distinct arrays always represent distinct storage. Arrays are in fact building blocks for slices.

Because slices are references, they don't use up additional memory and they are more efficient to use than arrays. That's the reason why they are used much more than arrays in Go-code. The format of the declaration is:

```
var identifier []type //no length is specified.
```

A slice that has not yet been initialized is set to *nil* by default and has length 0. Here is the format of the initialization of a slice:

```
var slice1 []type = arr1[start:end]
```

This represents the subarray of `arr1` (slicing the array, `start:end` is called a *slice-expression*) composed of the items from index `start` to index `end-1`. So the following statement is true:

the following statement is true:

```
slice1[0] == arr1[start]
```

This can be defined even before the array `arr1` is populated.

If one writes:

```
var slice1 []type = arr1[:]
```

then, `slice1` is equal to the complete array `arr1` (this is a shortcut for `arr1[0:len(arr1)]`). Another way to write this is:

```
slice1 = &arr1
```

`arr1[2:]` is the same as `arr1[2:len(arr1)]`, so it contains all the items of the array from the 2nd element until the last. Additionally, `arr1[:3]` is the same as `arr1[0:3]`, they both contain the array-items from the 1st until (not including) the 3rd element. If you need to cut the last element from `slice1`, use:

```
slice1 = slice1[:len(slice1)-1]
```

A slice of the array with elements 1, 2 and 3 can be made as follows:

```
s := [3]int{1,2,3}[:]
```

or

```
s := [...]int{1,2,3}[:]
```

or even shorter

```
s := []int{1,2,3}
```

This means that slices can also be initialized like arrays:

```
var x = []int{2, 3, 5, 7, 11}
```

`s2 := s[:]` is a slice made out of a slice, having identical elements, but it still refers to the same underlying array. A slice `s` can be expanded to its refers to the same underlying array. Hence `s` can be expanded to its

maximum size with `s = s[:cap(s)]`; any greater expansion gives a runtime error. For every slice (also for strings), the following is always true:

```
s == s[:i] + s[i:] // i is an int: 0 <= i <= len(s)
```

What this does is create an array of length 5 and then a slice to refer to it. Run the following program to see how slices work.

```go
package main
import "fmt"

func main() {
  var arr1 [6]int
  var slice1 []int = arr1[2:5] // item at index 5 not included!
  // load the array with integers: 0,1,2,3,4,5
  for i := 0; i < len(arr1); i++ {
    arr1[i] = i
  }
  // print the slice:
  for i := 0; i < len(slice1); i++ {
    fmt.Printf("Slice at %d is %d\n", i, slice1[i])
  }
  fmt.Printf("The length of arr1 is %d\n", len(arr1))
  fmt.Printf("The length of slice1 is %d\n", len(slice1))
  fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))
  // grow the slice:
  slice1 = slice1[0:4]
  for i := 0; i < len(slice1); i++ {
    fmt.Printf("Slice at %d is %d\n", i, slice1[i])
  }
  fmt.Printf("The length of slice1 is %d\n", len(slice1))
  fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))
  // grow the slice beyond capacity:
  // . slice1 = slice1[0:7] // panic: runtime error: slice bounds out of range

}
```

Arrays and Slices

Let's study the code above line by line. In main at **line 5**, we make an array `arr` of length **6**. Now, before even initializing the values of `arr`, we made a slice `slice1` out of `arr` at **line 6**: `var slice1 []int = arr1[2:5]`. The `slice1` contains three elements from `arr` (from index 2 to index 4). Then, we are initializing elements in the array `arr` with *for* loop at **line 8**. The element at index `i` is given the value of the iterator `i`. So `arr` holds **[0,1,2,3,4,5]**. The for index `i` is given the value of the iterator `i`. So `arr` holds **[0,1,2,3,4,5]**. The for loop at **line 12** is printing all the elements from the `slice1`. You'll note that

although we make `slice1` when values are not given to array `arr`, `slice1` still holds the value from 2nd index to 4th index of `arr`. So `slice1` is **[2,3,4]**.

Line 15 is printing the length of `arr`, which is **6**. **Line 16** is printing the length of `slice1`, which is **3** (as three elements are present), and **line 17** is printing the capacity of `slice1`, which is **4** (since three elements are present in `slice1` and only one element of `arr` is beyond the `slice1`).

At **line 19**, we are growing the `slice1` as: `slice1 = slice1[0:4]`. This means that the one last element of `arr1` which was not part of `slice1` initially, now is a part of `slice1`. The for loop at **line 20** is printing all the elements from the updated `slice1`. **Line 23** is printing the length of `slice1`, which is **4** (since four elements are present), and **line 24** is printing the capacity of `slice1`, which is **4** (since four elements are present in `slice1` and no element of `arr1` is beyond the `slice1`).

If `s2` is a slice, then you can move the slice forward by one with:

```
s2 = s2[1:]
```

But, the end is not moved. Slices cannot be re-sliced below zero to access earlier elements in the array, they can only move forward:

```
s2 = s2[-1:] // results in a compile error.
```

That's it about the brief introduction to slices in Golang. In the next lesson, you will learn about slices in further detail and learn how to handle slices within functions.