

The anatomy of Slices in Go

Slices are like Arrays but they can vary in length.



Uday Hiwarale

[Follow](#)

Sep 10, 2018 · 13 min read



What is a slice

A slice is like an array which is a **container to hold elements of the same data type** but slice can **vary in size**.

`slice` is a **composite data type** and because it is composed of primitive data type (see variables lesson for primitive data types).

Syntax to define a slice is pretty similar to that of an `array` **but without specifying the elements count**. Hence `s` is a slice

```
var s []int
```

Above code will create a slice of data type `int` that means it will hold elements of data type `int`. But what is a **zero-value** of a slice? As we saw in arrays, zero value of an `array` is an array with all its element being zero-value of data type it contains. Like an array of `int` with size `n` will have `n` zeroes as its elements because of zero value of `int` is `0`. But in the case of `slice`, zero value of slice defined like above is `nil`.

Below program will return `true`.

The screenshot shows a dark-themed instance of Visual Studio Code. In the Explorer sidebar, there is a tree view with 'OPEN EDITORS' expanded, showing 'slices.go' under 'src'. Below it, 'MAIN' is expanded, showing 'bin', 'pkg', and 'src' subfolders, with 'slices.go' also listed under 'src'. The main editor area displays the file 'slices.go' with the following content:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var s []int
7     fmt.Println(s == nil)
8 }
```

Below the editor is a terminal window titled 'TERMINAL' with the command 'go run src/slices.go' entered and the output 'true' displayed. The status bar at the bottom shows 'Ln 9, Col 1' and other standard terminal settings.

<https://play.golang.org/p/JI6ikCK2f9x>

But why `nil` though, you ask. Because **slice is just a reference to an array**.

`nil` or not, `slice` has type of `[]Type`. In above example, slice `s` has type of `[]int`.

☞ slice is a reference to an array

This may sound weird, but slice does not contain any data. It rather stores data in an array. But then you may ask, how that is even possible when `array` length is fixed?

`slice` when needed to store more data, creates a new `array` of appropriate length behind the scene to accommodate more data.

When a `slice` is created by simple syntax `var s []int`, it is not referencing any array, hence its value is `nil`. Let's now look at how it references an array.

Let's create an array and copy some of the element from that array to slice.

```

slices.go — main
1 package main
2
3 import "fmt"
4
5 func main() {
6     // define empty slice
7     var s []int
8
9     fmt.Println("s == nil", s == nil)
10
11    // create an array of int
12    a := [9]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
13
14    // creates new slice
15    s = a[2:4]
16    fmt.Println("s == nil", s == nil, "and s = ", s)
17
18 }

```

TERMINAL

```

main > go run src/slices.go
s == nil true
s == nil false and s = [3 4]
main >

```

https://play.golang.org/p/1naC_0qQz_E

In the above program, we have defined a slice `s` of type `int` but this slice doesn't reference an array. Hence, it is `nil` and first `Println` statement will print `true`.

Later, we created an array `a` of type `int` and assigned `s` with a new slice returned from `a[2:4]`. `a[2:4]` syntax returns a slice from the array `a` starting from `2` index element to `3` index element. I will explain `[:] operator` later.

Now, since `s` references array `a`, it must not be `nil` which is `true` from second `Println` and `s` is `[3,4]`.

Since, a `slice` always references an array, we can modify an array and check if that reflects in the `slice`.

In the above program, let's change the value of the **3rd** and **4th** element of the array `a` (`index 2 and 3 respectively`) and check the value of the slice `s`.

```

slices.go — main
1 package main
2
3 import "fmt"
4
5 func main() {
6     var s []int
7     a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
8     s = a[2:4]
9
10    a[2] = 33
11    a[3] = 44
12
13    fmt.Println(s)
14
15
TERMINAL
1: bash + □ □ □ □ □ □ □ □ □
main > go run src/slices.go
[33 44]
main > █

```

<https://play.golang.org/p/9xi8b8TTqHY>

From the above result, we are convinced that `slice` indeed is just a reference to an array and any change in that array will reflect in the slice.

Length and Capacity of a slice

As we have seen from the **array** lesson, to find of the length of a data type, we use `len` function. We are using the same `len` function for slices as well.

```

slices.go — main
1 package main
2
3 import "fmt"
4
5 func main() {
6     var s []int
7
8     a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
9
10    s = a[2:4]
11
12    fmt.Println("Length of s =", len(s))
13
14

```

```
TERMINAL
main > go run src/slices.go
Length of s = 2
main >
```

Ln 10, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

<https://play.golang.org/p/tKJaxdY7dYp>

The above program will print `Length of s = 2` on the screen which is correct because **it references only 2 elements from the array `a`.**

The **capacity** of a slice is the number of elements it can hold. Go provides a built-in function `cap` to get this capacity number.

```
slices.go — main
EXPLORER
OPEN EDITORS
  slices.go src
MAIN
  bin
  pkg
  src
    slices.go

slices.go ×
1 package main
2
3 import "fmt"
4
5 func main() {
6     var s []int
7
8     a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
9
10    s = a[2:4]
11
12    fmt.Println("Capacity of s =", cap(s))
13}
```

```
TERMINAL
main > go run src/slices.go
Capacity of s = 7
main >
```

Ln 13, Col 2 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

<https://play.golang.org/p/eAbelmHUK7K>

Above program returns `7` which is the **capacity** of the slice. Since `slice` references an array, it could have referenced array till the end. Since starting from the index `2` in the above example, there are `7` elements in the array, hence the capacity of the array is `7`.

Does that mean we can grow slice beyond its natural capacity? **Yes, you can.** We will find that out with `append` function.

slice is a struct

We will learn `struct` in upcoming lessons but `struct` is a `type` composed of different fields of different types from which variables of that struct-type is created.

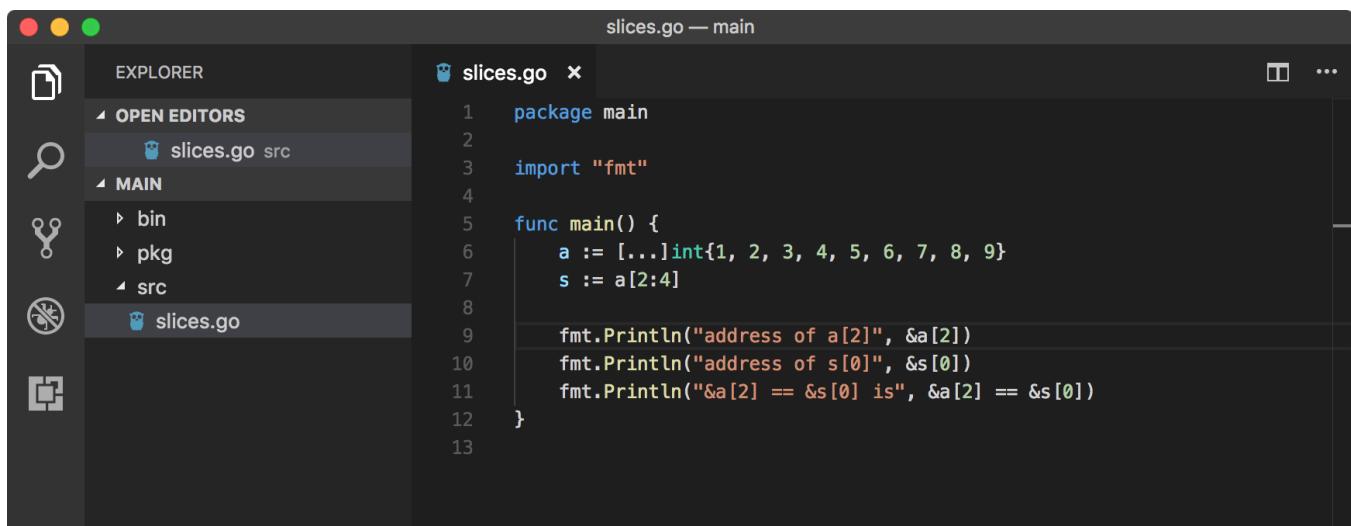
A `slice` struct looks like below

```
type slice struct {
    zerothElement *type
    len int
    cap int
}
```

A `slice` struct is composed of `zerothElement` pointer which points to the first element of an array that slice references. `len` and `cap` is the length and capacity of a slice respectively. `type` is the type of elements that underneath (*referenced*) array is composed of.

Hence when a new slice is defined, `zerothElement` pointer is set to its zero-value which is `nil`. But when a slice references an array, that pointer will not be `nil`.

We will learn more about `pointers` in upcoming lessons but the following example will show `address` of `a[2]` and `s[0]` is the same which means they are the exactly the same element in the memory.



```
package main
import "fmt"
func main() {
    a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
    s := a[2:4]
    fmt.Println("address of a[2]", &a[2])
    fmt.Println("address of s[0]", &s[0])
    fmt.Println("&a[2] == &s[0] is", &a[2] == &s[0])
}
```

```
TERMINAL
1: bash + - × ^ □ ×
main > go run src/slices.go
address of a[2] 0xc420018100
address of s[0] 0xc420018100
&a[2] == &s[0] is true
main > [REDACTED]
```

Ln 9, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing 😊 🔔

<https://play.golang.org/p/0jUjmjhTCos>

`0xc420018100` is a hexadecimal value of the memory location. You may get different results.

What will happen to the array if I change the value of an element in the slice? That is a very good question. As we know, slice doesn't hold any data, rather the data is held by an array. If we change some element values in the slice, that should reflect in the array.

slices.go — main

EXPLORER

- OPEN EDITORS
 - slices.go src
- MAIN
 - bin
 - pkg
 - src
 - slices.go

```
3 import "fmt"
4
5 func main() {
6     a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
7     s := a[2:4]
8     fmt.Println("before -> a[2] =", a[2])
9
10    s[0] = 33
11    fmt.Println("after -> a[2] =", a[2])
12 }
```

TERMINAL

```
1: bash + - × ^ □ ×
main > go run src/slices.go
before -> a[2] = 3
after -> a[2] = 33
main > [REDACTED]
```

Ln 11, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing 😊 🔔

<https://play.golang.org/p/eFChls0-66G>

☞ append function

You can append new values to the slice using built-in `append` function. Signature of `append` function is

```
func append(slice []Type, elems ...Type) []Type
```

Which means that `append` function takes a `slice` as the first argument, one/many elements as further arguments to append to the slice and returns a new slice of the same data type. Hence `slice` is a `variadic function` (*we will learn about variadic functions in upcoming lessons*).

Since `append` does not mutate original slice, let's see how it works.

```

slices.go — main
EXPLORER
OPEN EDITORS
  slices.go src
MAIN
  bin
  pkg
  src
    slices.go
slices.go x

1 package main
2
3 import "fmt"
4
5 func main() {
6     a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
7     s := a[2:4]
8     newS := append(s, 55, 66)
9
10    fmt.Printf("s=%v, newS=%v\n", s, newS)
11    fmt.Printf("len=%d, cap=%d\n", len(newS), cap(newS))
12    fmt.Printf("a=%v", a)
13 }
14

TERMINAL
1: bash
main > go run src/slices.go
s=[3 4], newS=[3 4 55 66]
len=4, cap=7
a=[1 2 3 4 55 66 7 8 9]main >

```

<https://play.golang.org/p/dSA5x7TkFeS>

As we can see from the above results, `s` remains unchanged and two new elements go copied to `newS` but look what happened to the array `a`. It got changed. `append` function mutated array referenced by slice `s`.

This is absolutely horrible. Hence **slices** are no easy business. Use `append` only to self assign the new slice like `s = append(s, ...)` which is more manageable.

What will happen if I append more elements than the capacity of a slice? Again, great question. How about we try it first.

The screenshot shows the RunGo IDE interface. The left sidebar displays the file structure under 'MAIN': 'bin', 'pkg', and 'src'. Inside 'src' is a folder 'slices.go' which contains a file 'slices.go'. The main editor window shows the code for 'slices.go':

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
7     s := a[2:4]
8     fmt.Printf("before -> s=%v\n", s)
9     fmt.Printf("before -> a=%v\n", a)
10    fmt.Printf("before -> len=%d, cap=%d\n", len(s), cap(s))
11    fmt.Println("&a[2] == &s[0] is", &a[2] == &s[0])
12
13    s = append(s, 50, 60, 70, 80, 90, 100, 110)
14    fmt.Printf("after -> s=%v\n", s)
15    fmt.Printf("after -> a=%v\n", a)
16    fmt.Printf("after -> len=%d, cap=%d\n", len(s), cap(s))
17    fmt.Println("&a[2] == &s[0] is", &a[2] == &s[0])
18 }

```

The terminal window below shows the output of running the program:

```

main > go run src/slices.go
before -> s=[3 4]
before -> a=[1 2 3 4 5 6 7 8 9]
before -> len=2, cap=7
&a[2] == &s[0] is true
after -> s=[3 4 50 60 70 80 90 100 110]
after -> a=[1 2 3 4 5 6 7 8 9]
after -> len=9, cap=14
&a[2] == &s[0] is false
main >

```

<https://play.golang.org/p/qKtVAka498Z>

So first we created an array `a` of `int` and initialized with a bunch of values. Then we created the slice `s` from array `a` starting from index `2` to `3`.

From the first set of `Print` statements, we verified values of `s` and `a`. Then we made sure that `s` references array `a` by matching memory address of their respective elements. Length and capacity of the slice `s` is also convincing.

Then we appended the slice `s` with `7` more values. So we expect the slice `s` to have `9` elements, hence its length is `9` but we have no idea about its new capacity. From later `Print` statement, we found that slice `s` got bigger than its initial capacity of `7` to `14` and its new length is `9`. But array `a` remain unchanged.

This looks weird at first but kinda amazing. Go figures out the math on its own that we are trying to push more values to the slice that its underneath array can't hold, so it creates a new array with greater length and copies old `slice` values to it. Then new

values from `append` is added to that array and origin array remain unchanged as no operation was done on it.

☞ anonymous array slice

Until now, we saw a slice which references array that we defined deliberately. But almost all the time, you would go with an array that is hidden and not accessible to the public.

Similar to an array, `slice` can be defined in a similar fashion with an initial value. In this case, go will create a hidden array to contain the values.

```
slices.go — main
slices.go x
1 package main
2
3 import "fmt"
4
5 func main() {
6     s := []int{1, 2, 3, 4, 5, 6}
7
8     fmt.Println("s=", s)
9     fmt.Printf("len=%d, cap=%d", len(s), cap(s))
10 }
11

TERMINAL
1: bash + - × ^ □ ×
main > go run src/slices.go
s= [1 2 3 4 5 6]
len=6, cap=6main > □
```

https://play.golang.org/p/L_uhlR5KjNY

It's pretty obvious that the capacity of this slice is `6` because the array is created by Go and Go preferred creating an array of length `6` as we are creating a slice of `6` elements. But what will happen when we append more two elements.

```
slices.go — main
slices.go x
1 package main
2
3 import "fmt"
4
5 func main() {
6     s := []int{1, 2, 3, 4, 5, 6}
```

```

    7 |     s = append(s, 7, 8)
    8 |
    9 |     fmt.Println("s=", s)
   10|     fmt.Printf("len=%d, cap=%d", len(s), cap(s))
   11|
   12}

TERMINAL
main > go run src/slices.go
s= [1 2 3 4 5 6 7 8]
len=8, cap=12main >

```

Ln 9, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

<https://play.golang.org/p/dmcnlc6Ys8c>

So, Go created an array of `12` length because when we are pushing `2` new elements to the slice, the original array of length `6` was not enough to hold `8` elements. No new array will be created if we appended new elements to the slice unless slice exceeds the length of `12`.

copy function

go provides built-in `copy` function to copy one slice into another. Signature of `copy` function is as below

```
func copy(dst, src []Type) int
```

Where `dst` is destination slice and `src` source slice. `copy` function will return the number of elements copied which is the minimum of `len(dst)` and `len(src)`.

```

slices.go — main
EXPLORER
OPEN EDITORS
slices.go src
MAIN
bin
pkg
src
slices.go

slices.go
package main
import "fmt"
func main() {
    var s1 []int
    s2 := []int{1, 2, 3}
    s3 := []int{4, 5, 6, 7}
    s4 := []int{1, 2, 3}

    n1 := copy(s1, s2)
    fmt.Printf("n1=%d, s1=%v, s2=%v\n", n1, s1, s2)
    fmt.Println("s1 == nil", s1 == nil)

    n2 := copy(s2, s3)

```

The screenshot shows the RunGo IDE interface. On the left is the code editor with the following Go code:

```

16   fmt.Printf("n2=%d, s2=%v, s3=%v\n", n2, s2, s3)
17
18   n3 := copy(s3, s4)
19   fmt.Printf("n3=%d, s3=%v, s4=%v\n", n3, s3, s4)
20 }
21

```

On the right is the terminal window showing the execution of the program:

```

main > go run src/slices.go
n1=0, s1=[], s2=[1 2 3]
s1 = nil true
n2=3, s2=[4 5 6], s3=[4 5 6 7]
n3=3, s3=[1 2 3 7], s4=[1 2 3]
main >

```

The status bar at the bottom indicates: Ln 14, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing.

<https://play.golang.org/p/MkFRM7I-v1R>

In the above program, we have defined `nil` slice `s1` and non-empty slices `s2` and `s3`. First `copy` statement tries to copy `s2` to `s1` but since `s1` is `nil` slice, nothing will happen and `s1` will be `nil`.

That won't be the case with `append`. As go is ready to create new array if needed, `append` on `nil` slice will work just fine.

In second `copy` statement, we are copying `s3` into `s2`, since `s3` contains 4 elements and `s2` contains 3 elements, only 3 (*min of 3 and 4*) will be copied. **Because `copy` does not append new elements.**

In third `copy` statement, we are copying `s4` into `s3`. Since `s4` contains 3 elements and `s4` contains 4, only 3 elements will be replaced in `s3`.

👉 make function

In the above example, we saw `s1` remain unchanged because it was `nil` slice. But there is a difference between `nil slice` and an `empty slice`. `nil` slice is a slice with **missing array reference** and `empty` slice is a slice with **empty array reference** or when the array is empty.

`make` is a built-in function that helps you create an empty slice. Signature of `make` function is as below. `make` function can create many empty composite types.

```
func make(t Type, size ...IntegerType) Type
```

In the case of `slice`, `make` function looks like below.

```
s := make([]type, len, cap)
```

Here, `type` is the data type of elements of a slice, `len` is the length of slice and `cap` is the capacity of the slice.

Let's try the previous example with `s1` being empty slice.

```
package main
import "fmt"
func main() {
    s1 := make([]int, 2, 4)
    s2 := []int{1, 2, 3}
    fmt.Printf("before => s1=%v, s2=%v\n", s1, s2)
    fmt.Println("before => s1 == nil", s1 == nil)

    n1 := copy(s1, s2)
    fmt.Printf("after => n1=%d, s1=%v, s2=%v\n", n1, s1, s2)
    fmt.Println("after => s1 == nil", s1 == nil)
}
```

TERMINAL

```
main > go run src/slices.go
before => s1=[0 0], s2=[1 2 3]
before => s1 == nil false
after => n1=2, s1=[1 2], s2=[1 2 3]
after => s1 == nil false
main > 
```

https://play.golang.org/p/z0tIrlRYI_hMu

Above result proves that empty slice was created and `copy` function does not append values to the slice beyond its length even when its capacity is larger.

Type... unpack operator

Some people call it `unpack` operator or `expand` operator, to me `spread` seems more natural. If you see `append` function syntax, it accepts more than one arguments to append elements to a slice. What if you have a slice and you need to append values

from it to another slice. In that case `...` operator is useful because `append` does not accept slice as an argument, only the type which slice element is made of.

```

slices.go — main
1 package main
2
3 import "fmt"
4
5 func main() {
6     s1 := make([]int, 0, 10)
7     fmt.Println("before -> s1=", s1)
8
9     s2 := []int{1, 2, 3}
10    s1 = append(s1, s2...)
11    fmt.Println("after -> s1=", s1)
12 }
13

```

TERMINAL

```

main > go run src/slices.go
before -> s1= []
after -> s1= [1 2 3]
main >

```

<https://play.golang.org/p/JflgynyqVYc>

☞ [start:end] extract operator

Go provides an amazing operator `[start:end]` (*I like to call it **extract** operator*) which you can use easily to extract any part of a slice. Both `start` and `end` are optional indexes. `start` is an initial index of slice while `end` is the last index up to which elements should be extracted hence `end` index is not included. **This syntax returns a new slice.**

```

slices.go — main
1 package main
2
3 import "fmt"
4
5 func main() {
6     s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
7
8     fmt.Println("s[:]", s[:])
9     fmt.Println("s[2:]", s[2:])
10    fmt.Println("s[:4]", s[:4])
11    fmt.Println("s[2:4]", s[2:4])
12 }
13

```

```

main > go run src/slices.go
s[:] [0 1 2 3 4 5 6 7 8 9]
s[2:] [2 3 4 5 6 7 8 9]
s[:4] [0 1 2 3]
s[2:4] [2 3]
main >

```

Ln 8, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing ☺ 🔔

<https://play.golang.org/p/lNhNx5KCVrR>

In the above example, we have the simple slice `s` of integers starting from `0` to `9`.

- `s[:]` means extract all elements of `s` starting from `0` index till the end. Hence returns all elements of `s`.
- `s[2:]` means extract elements of `s` starting from 2nd index till the end. Hence returns `[2 3 4 5 6 7 8 9]`
- `s[:4]` means extract elements of `s` starting from `0`th index till `4`th index but not including index `4`. Hence returns `[0 1 2 3]`
- `s[2:4]` means extract elements of `s` starting from 2nd index till `4`th index but not including index `4`. Hence returns `[2 3]`

The important thing to remember is that, any slice created by extract operator still references the same underneath array. You can use `copy`, `make` or `append` functions in conjugation to avoid this.

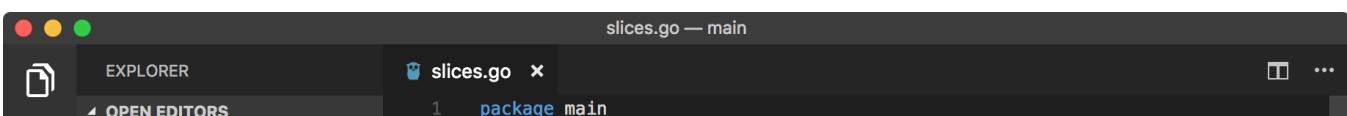
Golang also provides a mean to set capacity of

☞ Slice iteration

There is no difference as such between `array` and `slice` when it comes to iteration. Virtually, a `slice` is like an `array` with the same structure, you can use all the functionality of `array` while iterating over slices.

☞ Passed by reference

Well, slices are still passed by value to the function but since they reference the same underneath the array, it looks like that they are passed by reference.



```

2 import "fmt"
3
4 func makeSquares(slice []int) {
5     for index, elem := range slice {
6         slice[index] = elem * elem
7     }
8 }
9
10 func main() {
11     s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
12
13     makeSquares(s)
14
15     fmt.Println(s)
16 }
17
18

```

TERMINAL

```

main ▶ go run src/slices.go
[0 1 4 9 16 25 36 49 64 81]
main ▶

```

Ln 8, Col 6 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing

<https://play.golang.org/p/p6O0Uqewwlg>

In the above example, we have defined `makeSquares` which takes a slice and replaces elements of input slice with their squares. This will yield the following result

[0 1 4 9 16 25 36 49 64 81]

This proves that even though `slice` is passed by value, since it references the same underneath `array`, and we can change the value of the elements in that array.

Why we are so sure that `slice` is passed by value, change `makeSquares` function to `func makeSquares(slice []int) {slice = slice[1:5]}` which does not change `s` in the main function.

Let's see what will happen if we use the above program with `array` as an input argument to the function.

```

1 package main
2
3 import "fmt"
4
5 func makeSquares(array [10]int) {
6     for index, elem := range array {
7         array[index] = elem * elem
8     }
9 }
10
11 func main() {
12     s := [10]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
13
14     makeSquares(s)
15
16     fmt.Println(s)
17 }
18

```

The screenshot shows the RunGo IDE interface. On the left, the file tree shows a package named 'pkg' with a 'src' directory containing a file named 'slices.go'. The code editor on the right contains the following Go code:

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
7
8     // makeSquares receives a copy of s
9     makeSquares(s)
10
11    fmt.Println(s)
12 }

```

Below the code editor is a terminal window titled 'TERMINAL' showing the output of running the program:

```

main > go run src/slices.go
[0 1 2 3 4 5 6 7 8 9]
main >

```

The status bar at the bottom indicates 'Ln 12, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing'.

<https://play.golang.org/p/qF8grYQ8Q0s>

Above program will result into [0 1 2 3 4 5 6 7 8 9] which means `makeSquares` received only copy of it.

➡ Delete slice element(s)

go does not provide any keyword or function to delete `slice` elements directly. We need to use some hacks to get there. As deleting an element from a slice is like joining slice behind and ahead of the element which needs to be deleted, let's see how that works.

The screenshot shows the VS Code IDE interface. On the left, the file tree shows a package named 'main' with a 'src' directory containing a file named 'slices.go'. The code editor on the right contains the following Go code:

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
7
8     // delete element at index 2 (== 2)
9     s = append(s[:2], s[3:]...)
10
11    fmt.Println(s)
12 }

```

Below the code editor is a terminal window titled 'TERMINAL' showing the output of running the program:

```

main > go run src/slices.go
[0 1 3 4 5 6 7 8 9]
main >

```

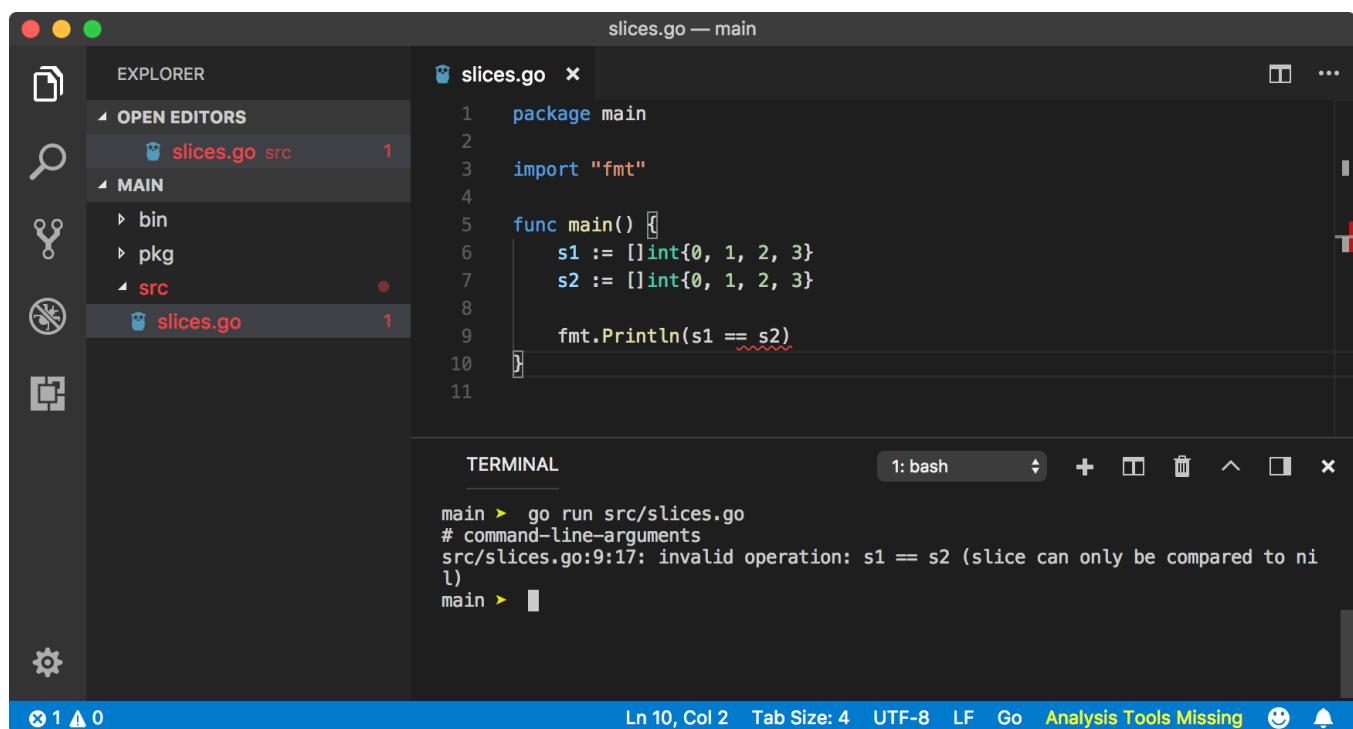
The status bar at the bottom indicates 'Ln 8, Col 1 Tab Size: 4 UTF-8 LF Go Analysis Tools Missing'.

<https://play.golang.org/p/IflGN2m-uSm>

In the above program, we have extracted a slice from `s` starting from index `0` up to but not including index `2` and appended with slice starting from index `3` till the end. This will create a new slice without index `2`. The above program will print `[0 1 3 4 5 6 7 8 9]`. Using this same technique, we can remove multiple elements from anywhere in the slice.

slice comparison

If you try the following program



The screenshot shows the GoLand IDE interface. On the left is the Explorer sidebar with 'OPEN EDITORS' containing 'slices.go' and 'src/slices.go'. The main area shows the code for 'slices.go':

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     s1 := []int{0, 1, 2, 3}
7     s2 := []int{0, 1, 2, 3}
8
9     fmt.Println(s1 == s2)
10}

```

Below the code editor is a terminal window titled 'TERMINAL' showing the output of running the program:

```

main > go run src/slices.go
# command-line-arguments
src/slices.go:9:17: invalid operation: s1 == s2 (slice can only be compared to nil)
main >

```

<https://play.golang.org/p/kZ7-SyCBvpt>

You will get error `invalid operation: s1 == s2 (slice can only be compared to nil)` which means that slices can be only checked for the condition of `nil` or not. If you really need to compare two slices, use `for range` loop to match each element of the two slices or use `DeepEqual` function of `reflect` package.

Multi-dimensional slices

Similar to `array`, slices can also be multi-dimensional. Syntax of defining multi-dimensional slices are pretty similar to arrays but without mentioning element size.

```
s1 := [][]int{
    []int{1, 2},
    []int{3, 4},
    []int{5, 6},
}

s2 := [][]int{
    {1, 2},
    {3, 4},
    {5, 6},
}
```

☞ Memory optimization

As we know, slice references an array. If there is a function that returns a `slice`, that slice might reference an array that is big in size. As long as that slice is in memory, the array cannot be garbage collected and will hold a large part of system memory.

Below is a bad program

```
package main

import "fmt"

func getCountries() []string {
    countries := []string{"United states", "United kingdom",
    "Australia", "India", "China", "Russia", "France", "Germany",
    "Spain"} // can be much more

    return countries[:3]
}

func main() {
    countries := getCountries()

    fmt.Println(cap(countries)) // 9
}
```

As you see, the capacity of the countries is `9` means underneath array holds `9` element (*we know in this case*).

To avoid that, we must create a new slice of an anonymous array which will be manageable in length. The following program is a good program.

```
package main

import "fmt"

func getCountries() []string {
    countries := []string{"United states", "United kingdom",
    "Australia", "India", "China", "Russia", "France", "Germany",
    "Spain"} // can be much more

    c = make([]string, 3) // made empty of length and capacity 3
    copy(c, countries[:3]) // copied to `c`

    return
}

func main() {
    countries := getCountries()

    fmt.Println(cap(countries)) // 3
}
```

☞ Read more

go does not provide fancy functions and methods like `JavaScript` to manipulate slices, as you saw. We used hacks in order to delete slice element(s). If you are looking for such hacks for fancy functions like pop, push, shift etc., follow <https://github.com/golang/go/wiki/SliceTricks>