The anatomy of Slices in Go

medium.com/rungo/the-anatomy-of-slices-in-go-6450e3bb2b94

Slices are like Arrays but they can vary in length.

What is a slice

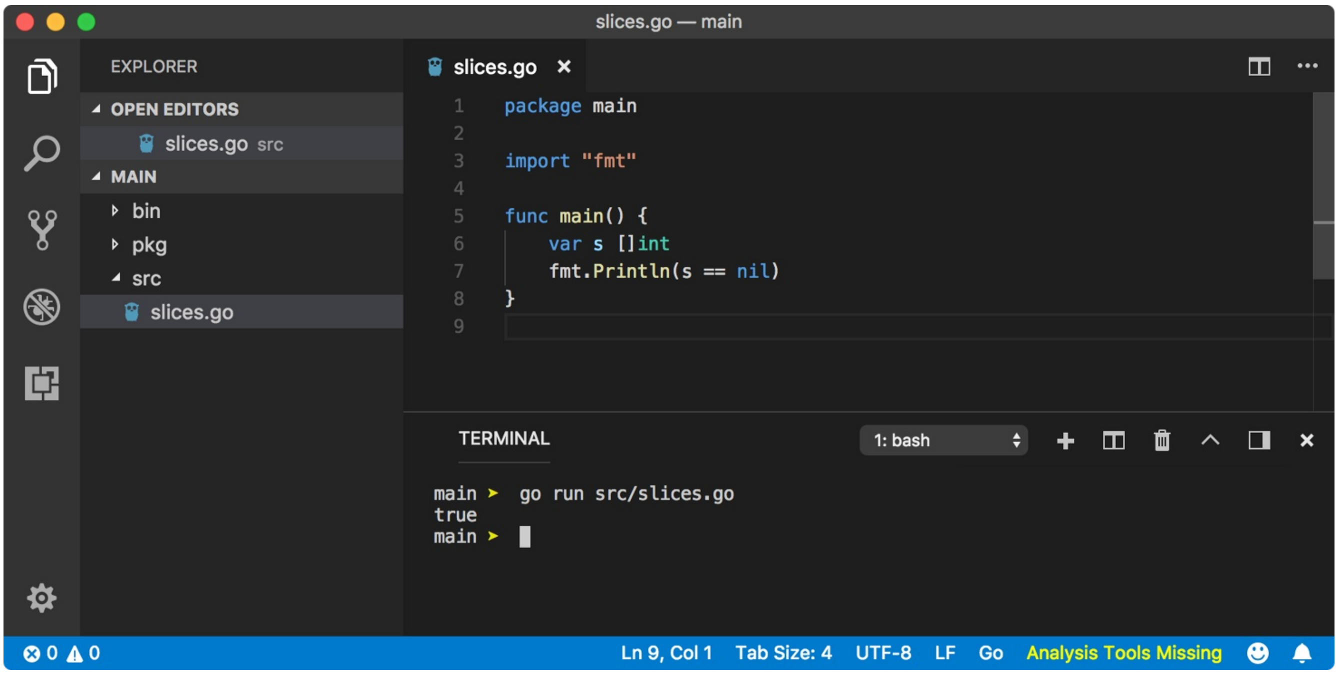
A slice is like an array which is a **container to hold elements of same data type** but slice can **vary in size**.

> `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 an an 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 zero value of `int` is `0`. But in case of `slice`, zero value of slice defined like above is `nil`. Below program will return `true`.



```go
package main

import "fmt"

func main() {
    var s []int
    fmt.Println(s == nil)
}
```

```
main ➤  go run src/slices.go
true
main ➤
```

https://play.golang.org/p/Jl6ikCK2f9x

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`*.*

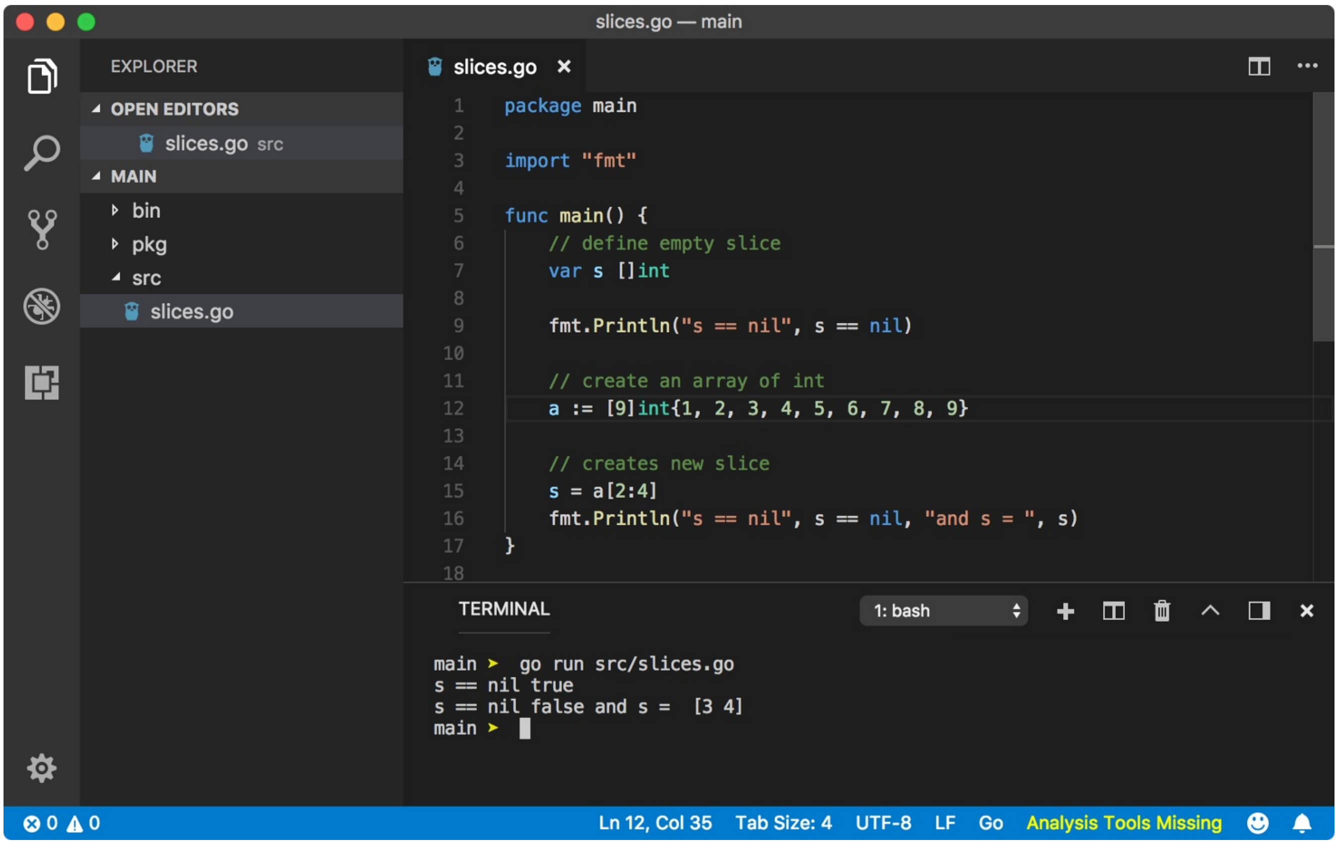
☛ `slice` is an reference to array

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.



```go
package main

import "fmt"

func main() {
	// define empty slice
	var s []int

	fmt.Println("s == nil", s == nil)

	// create an array of int
	a := [9]int{1, 2, 3, 4, 5, 6, 7, 8, 9}

	// creates new slice
	s = a[2:4]
	fmt.Println("s == nil", s == nil, "and s = ", s)
}
```

```
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

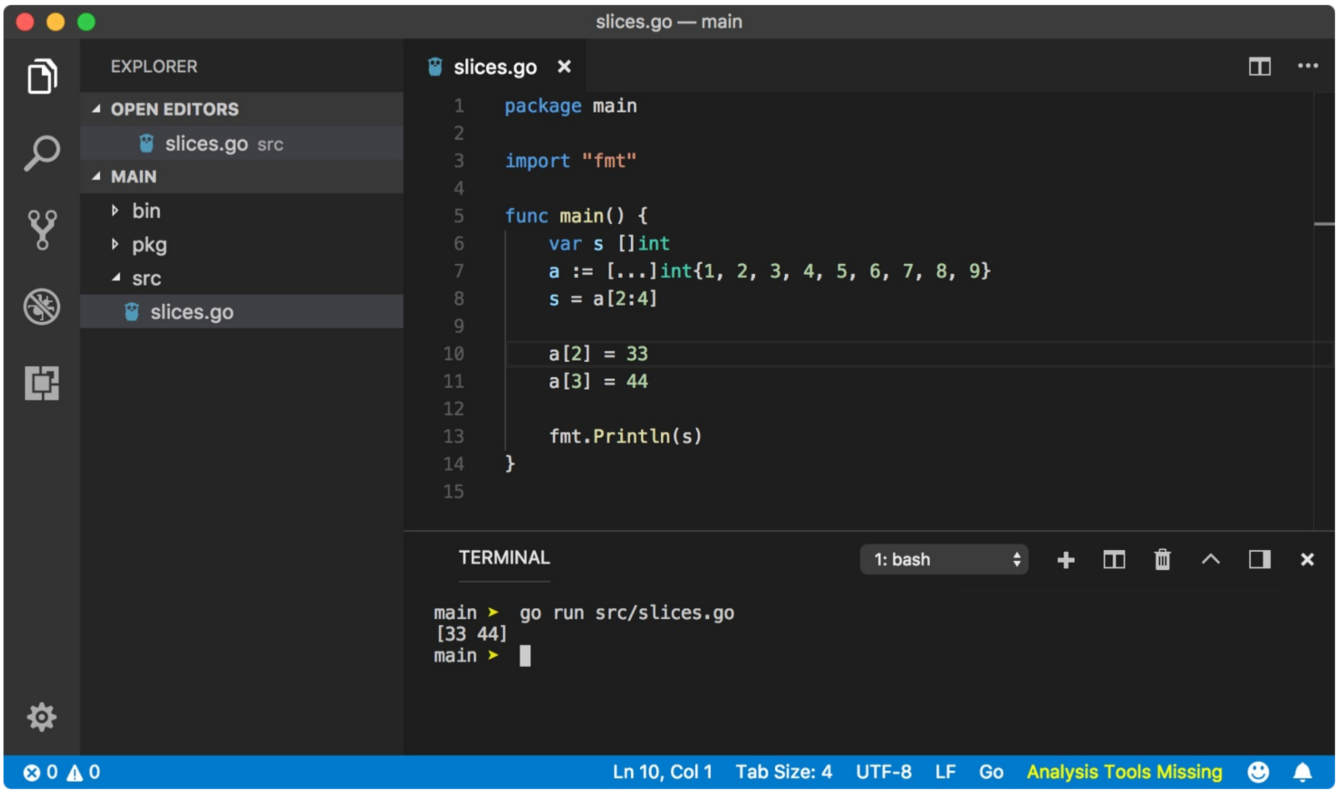
In above program, we have defined a slice `s` of type `int` but this slice doesn't reference any array. Hence, it is `nil` and first `Println` statement will print `true`.

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



```go
package main

import "fmt"

func main() {
    var s []int
    a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
    s = a[2:4]

    a[2] = 33
    a[3] = 44

    fmt.Println(s)
}
```

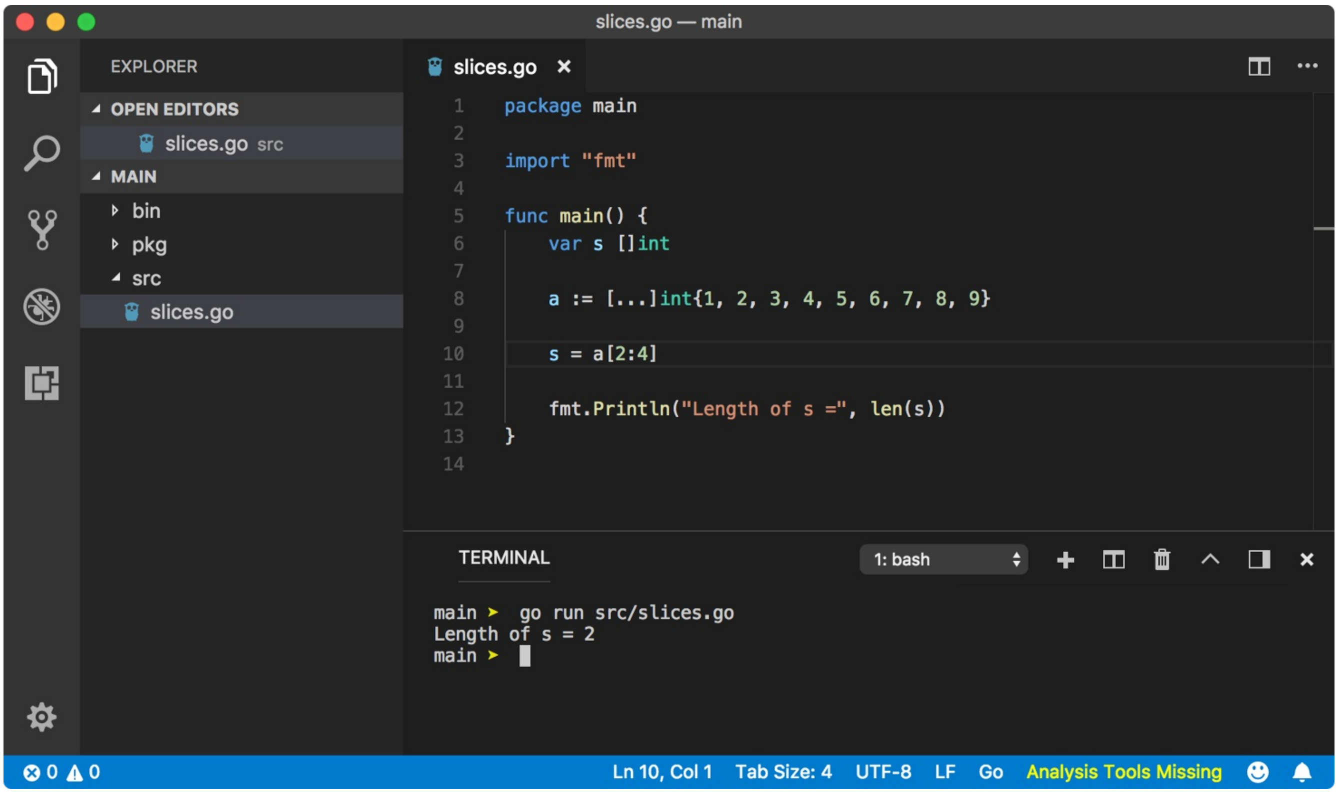
```
main ➤  go run src/slices.go
[33 44]
main ➤
```

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

From 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 length of a data type, we use `len` function. We are using same `len` function for slices as well.



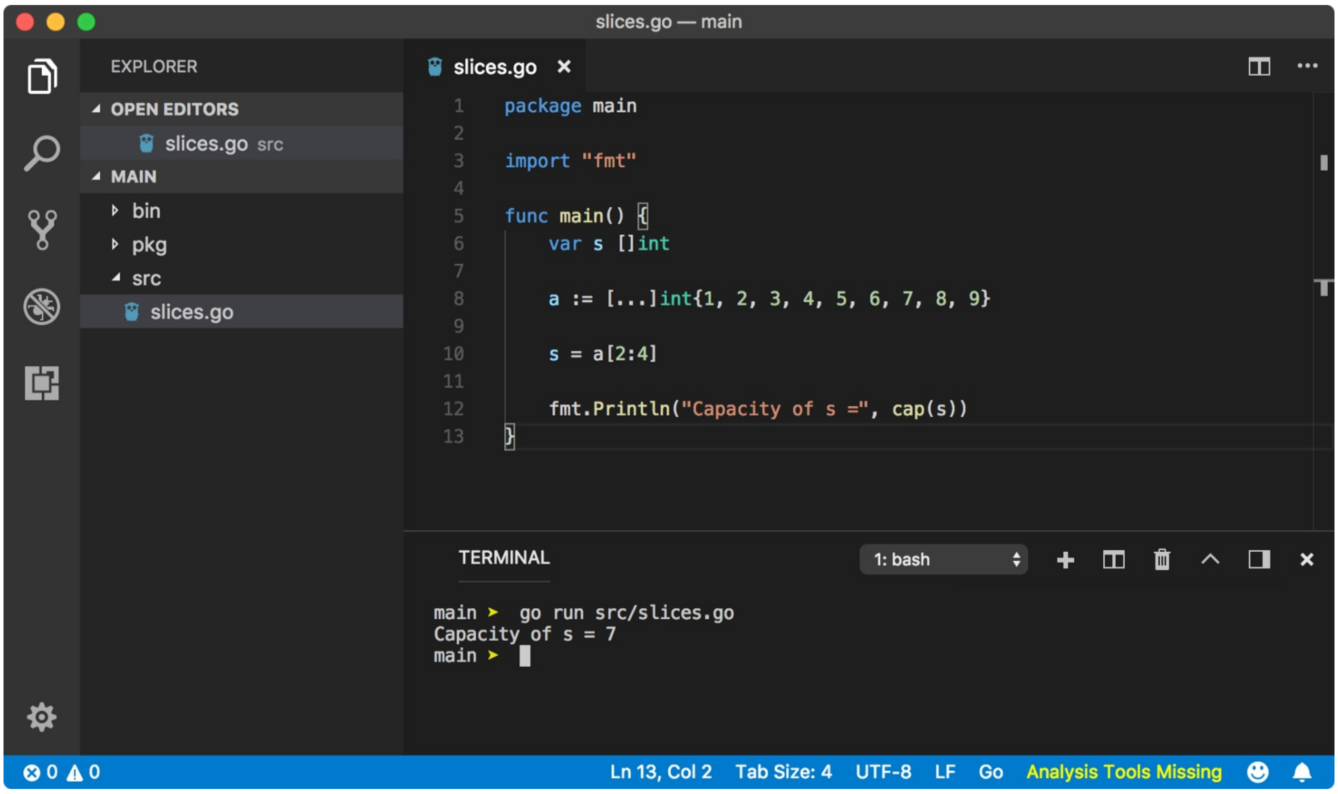
```go
package main

import "fmt"

func main() {
	var s []int

	a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}

	s = a[2:4]

	fmt.Println("Length of s =", len(s))
}
```

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

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

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

Capacity of a slice is number of elements it can hold. go provides built-in function `cap` to get this capacity number.



```go
package main

import "fmt"

func main() {
    var s []int

    a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}

    s = a[2:4]

    fmt.Println("Capacity of s =", cap(s))
}
```

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

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

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

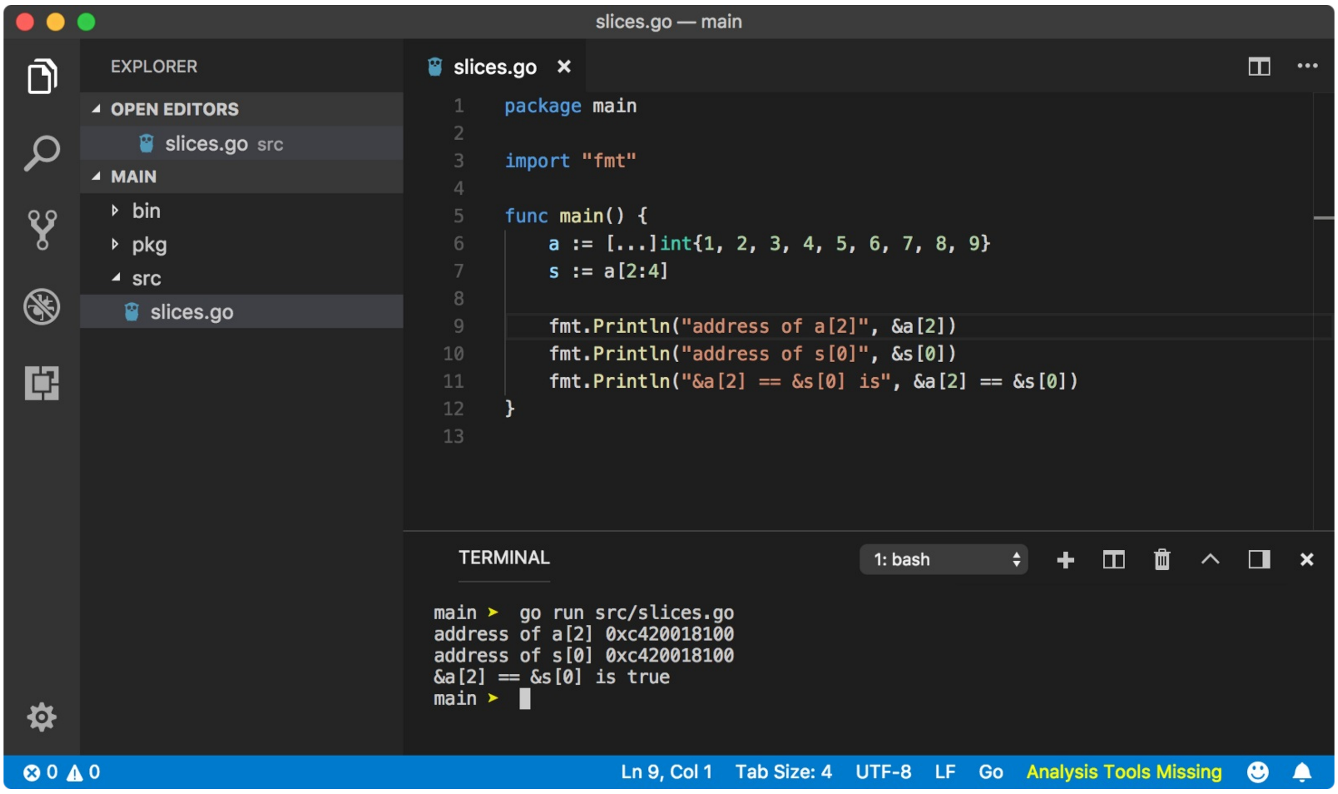
A `slice` struct looks like below

```
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 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 following example will show `address` of `a[2]` and `s[0]` is same which means they are exactly same element in the memory.



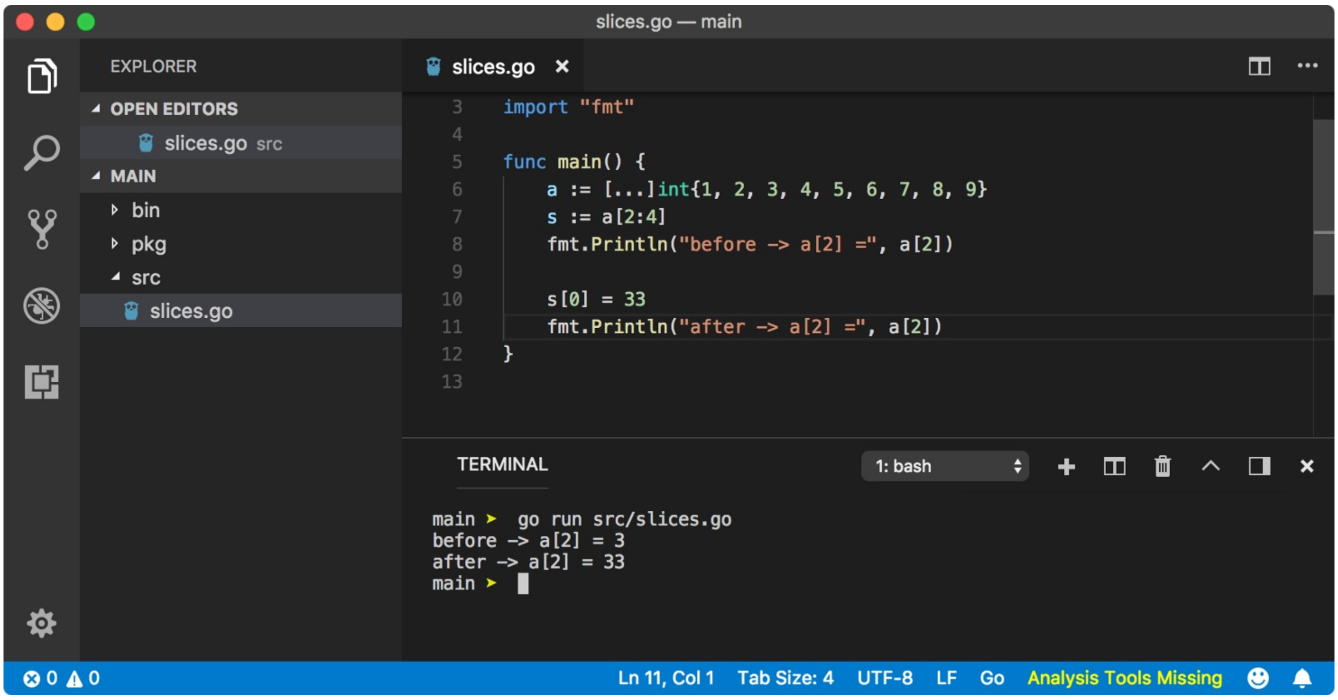
```go
package main

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])
}
```

```
main ➤  go run src/slices.go
address of a[2] 0xc420018100
address of s[0] 0xc420018100
&a[2] == &s[0] is true
main ➤
```

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 element in slice?That is a very good question. As we know, slice doesn't hold any data, rather data is held by the array. If we change some element values in slice, that should reflect in the array.



```go
import "fmt"

func main() {
    a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
    s := a[2:4]
    fmt.Println("before -> a[2] =", a[2])

    s[0] = 33
    fmt.Println("after -> a[2] =", a[2])
}
```

```
main ➤  go run src/slices.go
before -> a[2] = 3
after -> a[2] = 33
main ➤
```

https://play.golang.org/p/eEChIs0-66G

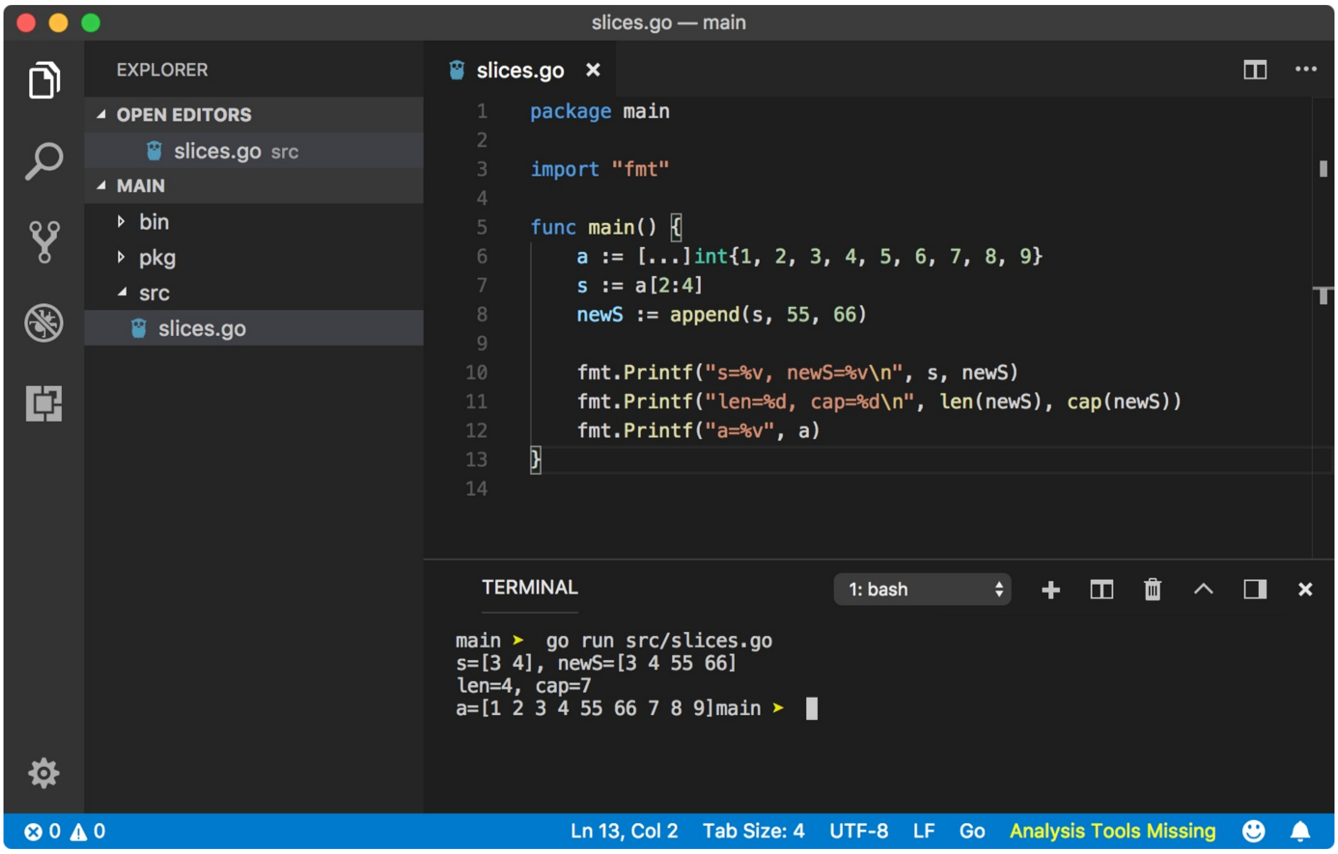
☛ `append` function

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 first argument, one/many elements as further arguments to append to the slice and returns a new slice of 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.



```go
package main

import "fmt"

func main() {
    a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
    s := a[2:4]
    newS := append(s, 55, 66)

    fmt.Printf("s=%v, newS=%v\n", s, newS)
    fmt.Printf("len=%d, cap=%d\n", len(newS), cap(newS))
    fmt.Printf("a=%v", a)
}
```

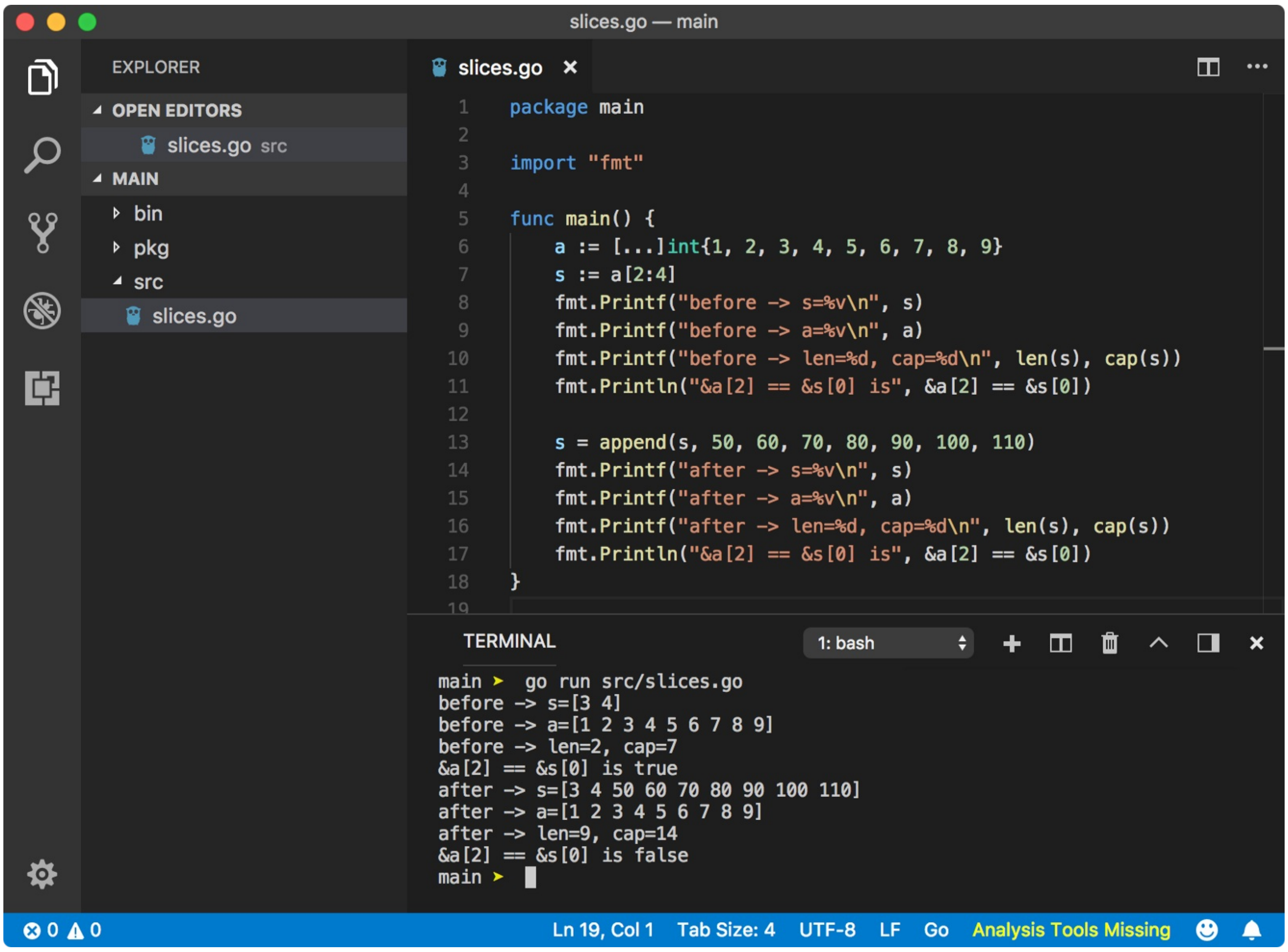
```
main ➤  go run src/slices.go
s=[3 4], newS=[3 4 55 66]
len=4, cap=7
a=[1 2 3 4 55 66 7 8 9]main ➤
```

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

As we can see from above results, `s` remains unchanged and two new elements go copied to `newS` but look what happened to 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.



```go
package main

import "fmt"

func main() {
	a := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9}
	s := a[2:4]
	fmt.Printf("before -> s=%v\n", s)
	fmt.Printf("before -> a=%v\n", a)
	fmt.Printf("before -> len=%d, cap=%d\n", len(s), cap(s))
	fmt.Println("&a[2] == &s[0] is", &a[2] == &s[0])

	s = append(s, 50, 60, 70, 80, 90, 100, 110)
	fmt.Printf("after -> s=%v\n", s)
	fmt.Printf("after -> a=%v\n", a)
	fmt.Printf("after -> len=%d, cap=%d\n", len(s), cap(s))
	fmt.Println("&a[2] == &s[0] is", &a[2] == &s[0])
}
```

```
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 bunch of values. Then we created slice `s` from array `a` starting from index `2` to `3`.

From first set of 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 slice `s` is also convincing.

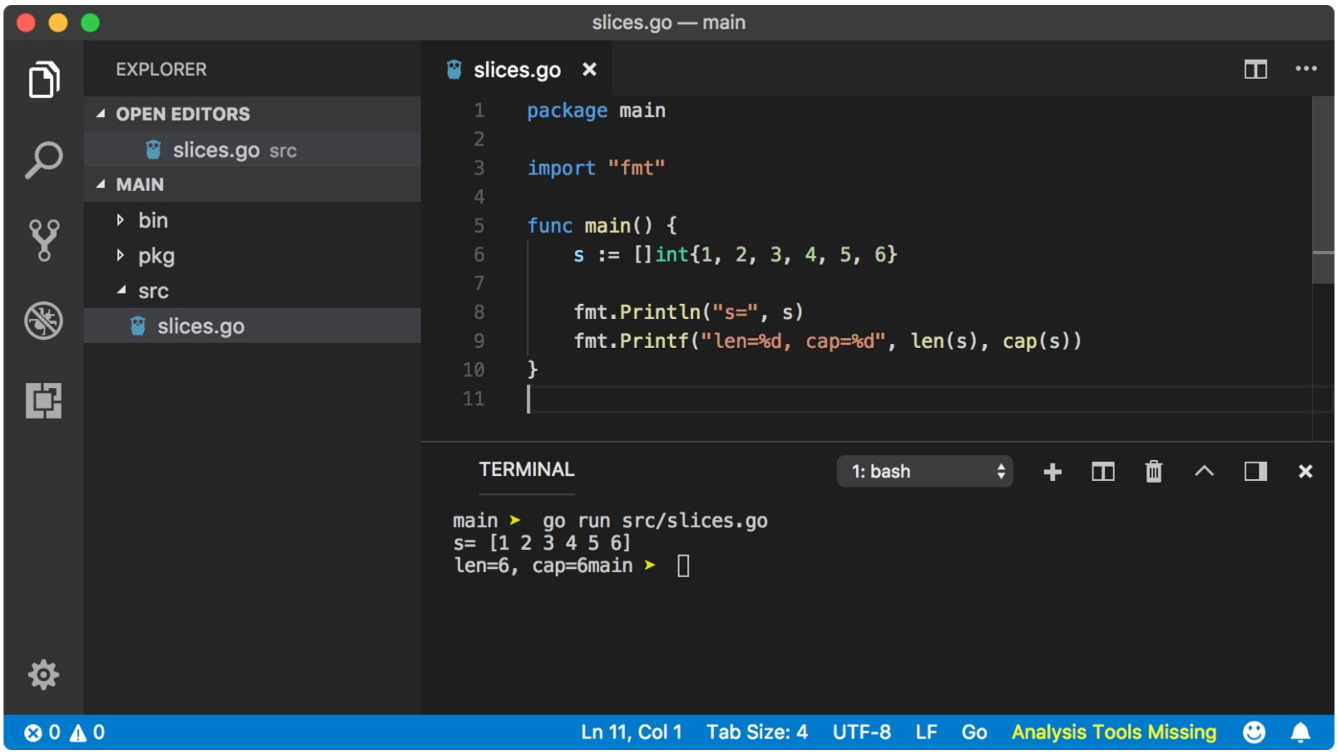
Then we appended slice `s` with `7` more values. So we expect slice `s` to have `9` elements, hence its length is `9` but we have no idea about its new capacity. From later statement, we found that slice `s` got bigger than its initial capacity of `7` to `14` and its new length is `9`. But array `a`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

This looks weird at first but kinda amazing. go figures out 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 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 remains unchanged as no operation was done on it.

☛ `anonymous array` slice

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

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

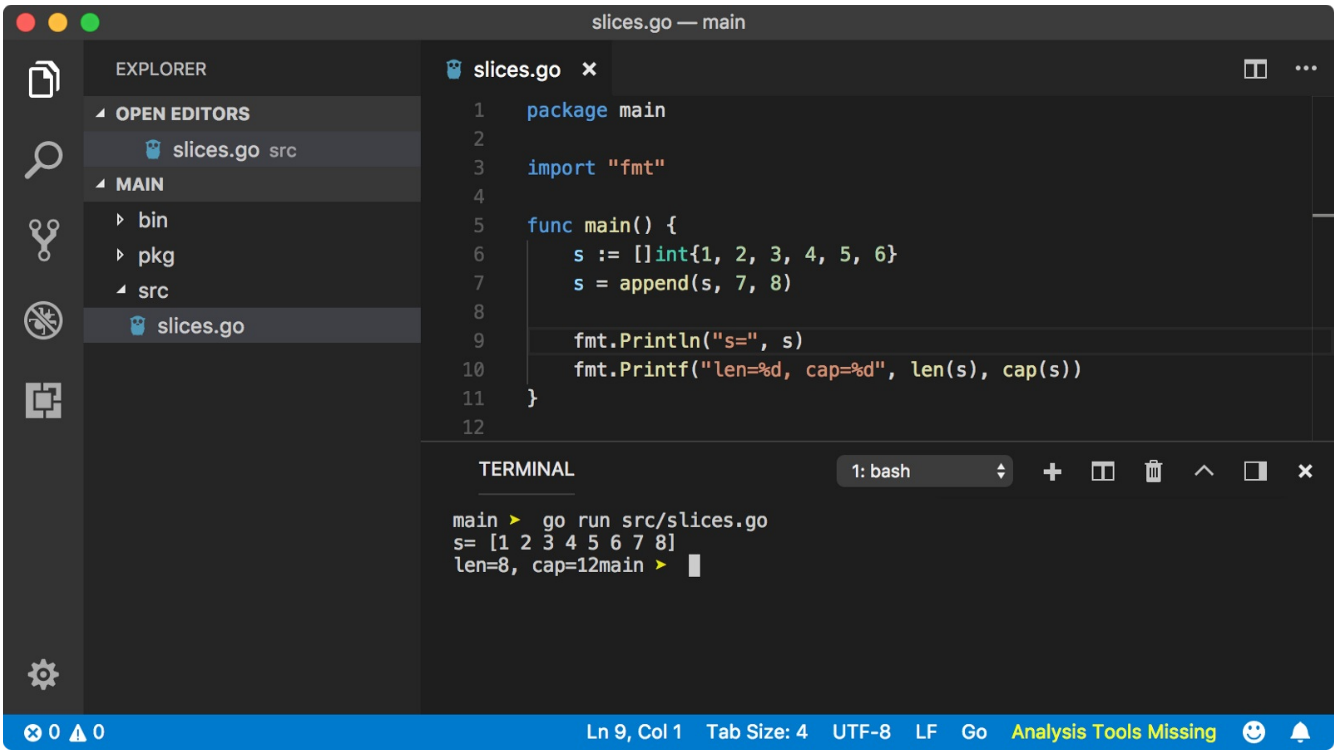


```go
package main

import "fmt"

func main() {
    s := []int{1, 2, 3, 4, 5, 6}

    fmt.Println("s=", s)
    fmt.Printf("len=%d, cap=%d", len(s), cap(s))
}
```

```
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 capacity of this slice is `6` because array is created by go and go preferred creating array of length `6` as we are creating slice of `6` elements. But what will happen when we append more two elements.



```go
package main

import "fmt"

func main() {
	s := []int{1, 2, 3, 4, 5, 6}
	s = append(s, 7, 8)

	fmt.Println("s=", s)
	fmt.Printf("len=%d, cap=%d", len(s), cap(s))
}
```

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

https://play.golang.org/p/dmcnLc6Ys8c

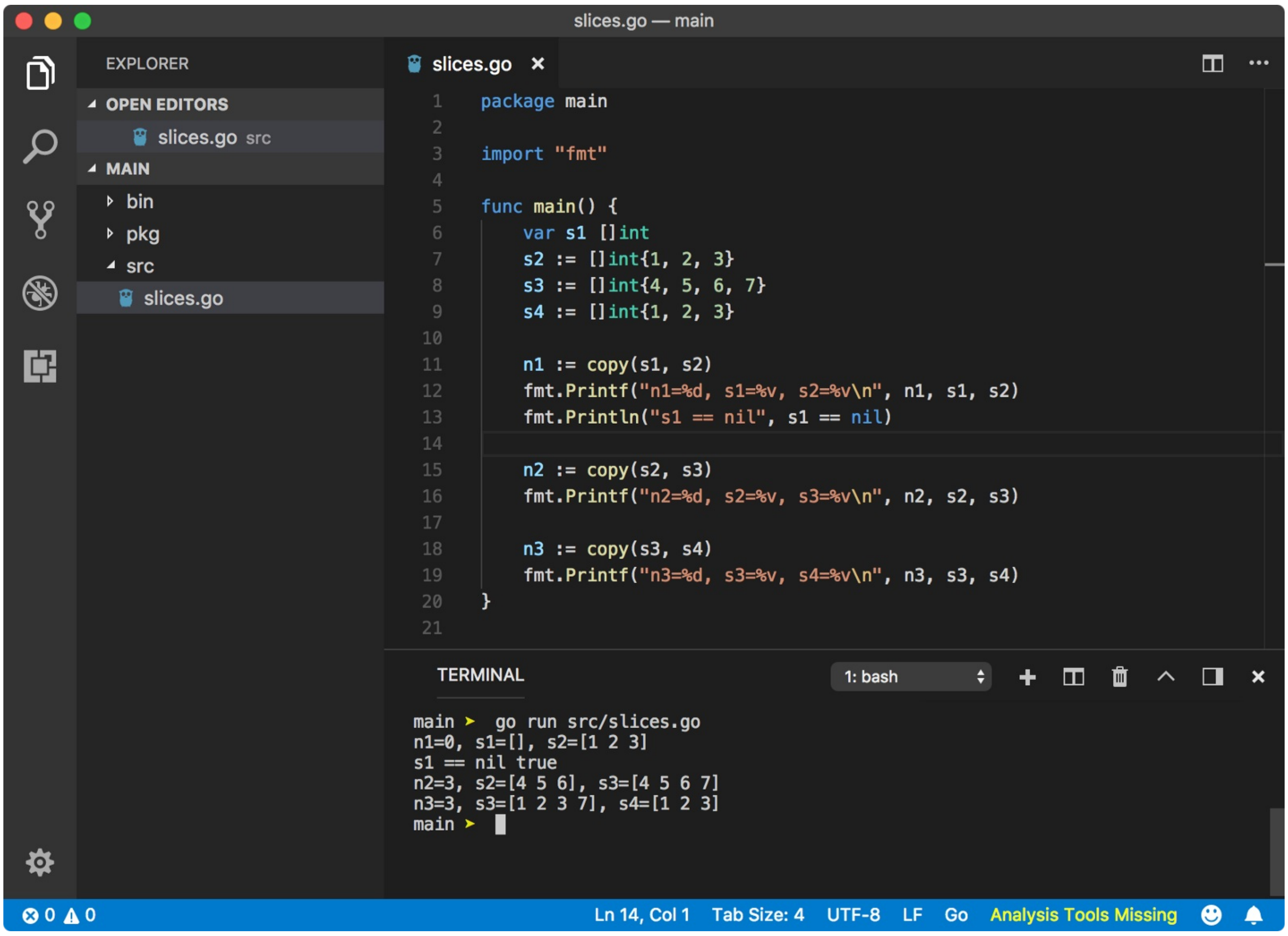
So, go created array of `12` length because when we are pushing `2` new elements to the slice, original array of length `6` was not enough to hold `8` elements. No new array will be created if we appended new elements to the slice unless slice exceed length of `12`.

☛ `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 number of elements copied which is minimum of `len(dst)` and `len(src)`.



```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)
    fmt.Printf("n2=%d, s2=%v, s3=%v\n", n2, s2, s3)

    n3 := copy(s3, s4)
    fmt.Printf("n3=%d, s3=%v, s4=%v\n", n3, s3, s4)
}
```

```
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 ➤
```

https://play.golang.org/p/MkFRMZl-v1B

In 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 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 array is empty.

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

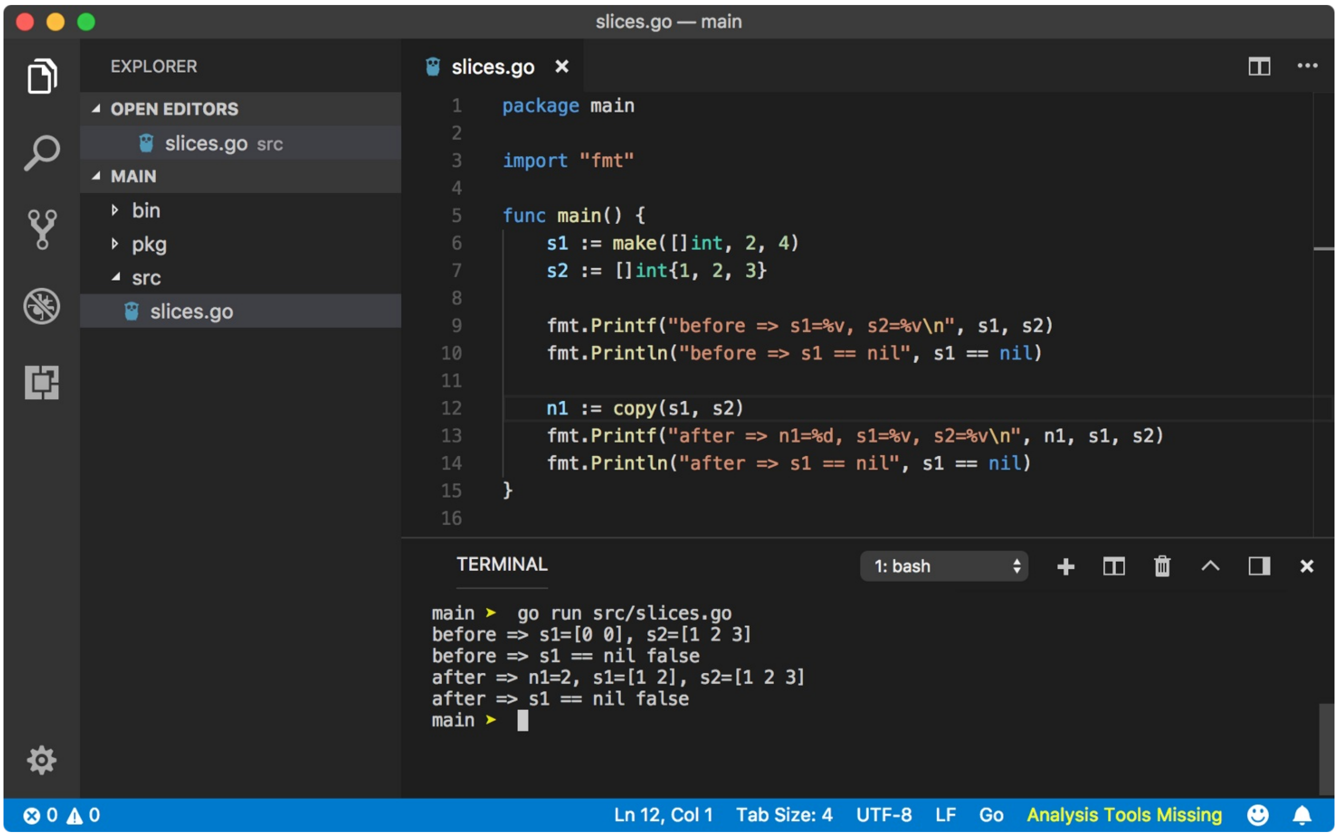
func make(t Type, size ...IntegerType) Type

In 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 length of slice and `cap` is capacity of the slice.

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



```go
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)
}
```

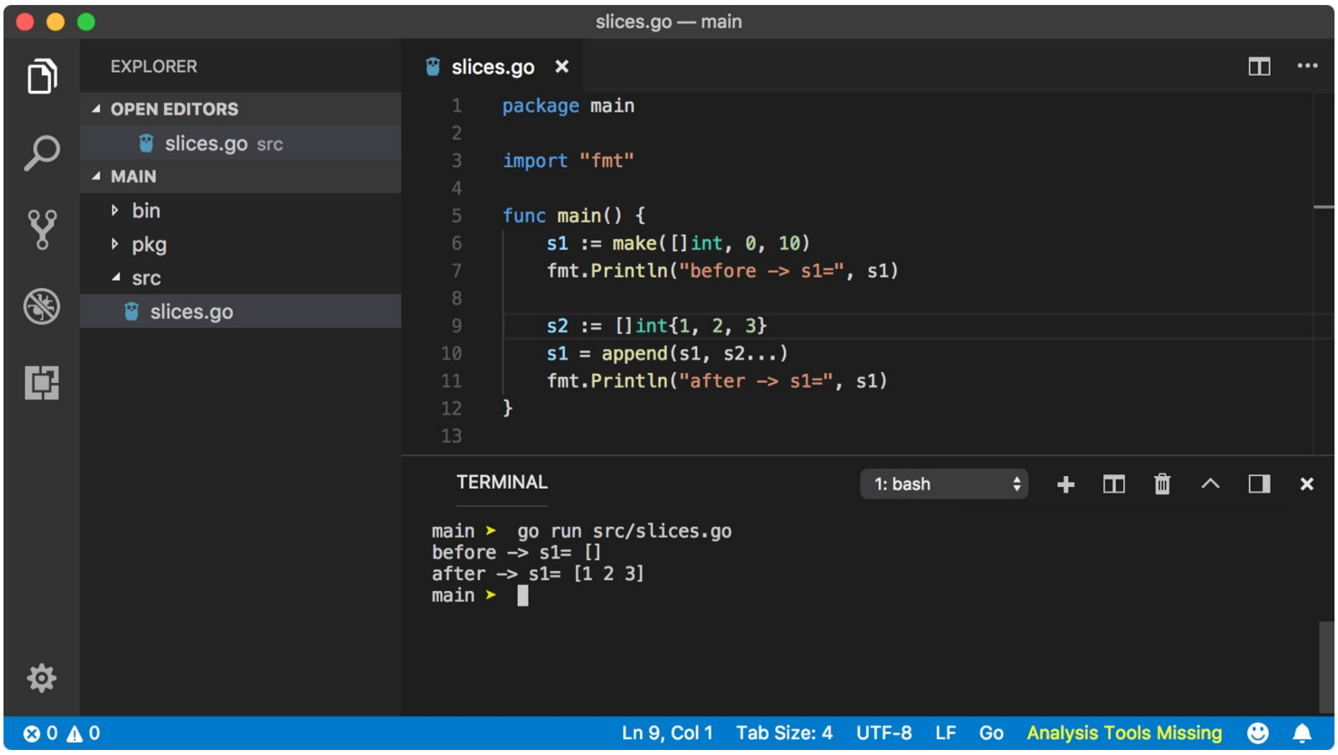
```
main ➤  go run src/slices.go
before => s1=[0 0], s2=[1 2 3]
before => s1 == nil false
after => n1=2, s1=[1 2], s2=[1 2 3]
after => s1 == nil false
main ➤
```

https://play.golang.org/p/z0tlrRYLhMu

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 is `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 a argument, only the type which slice element is made of.



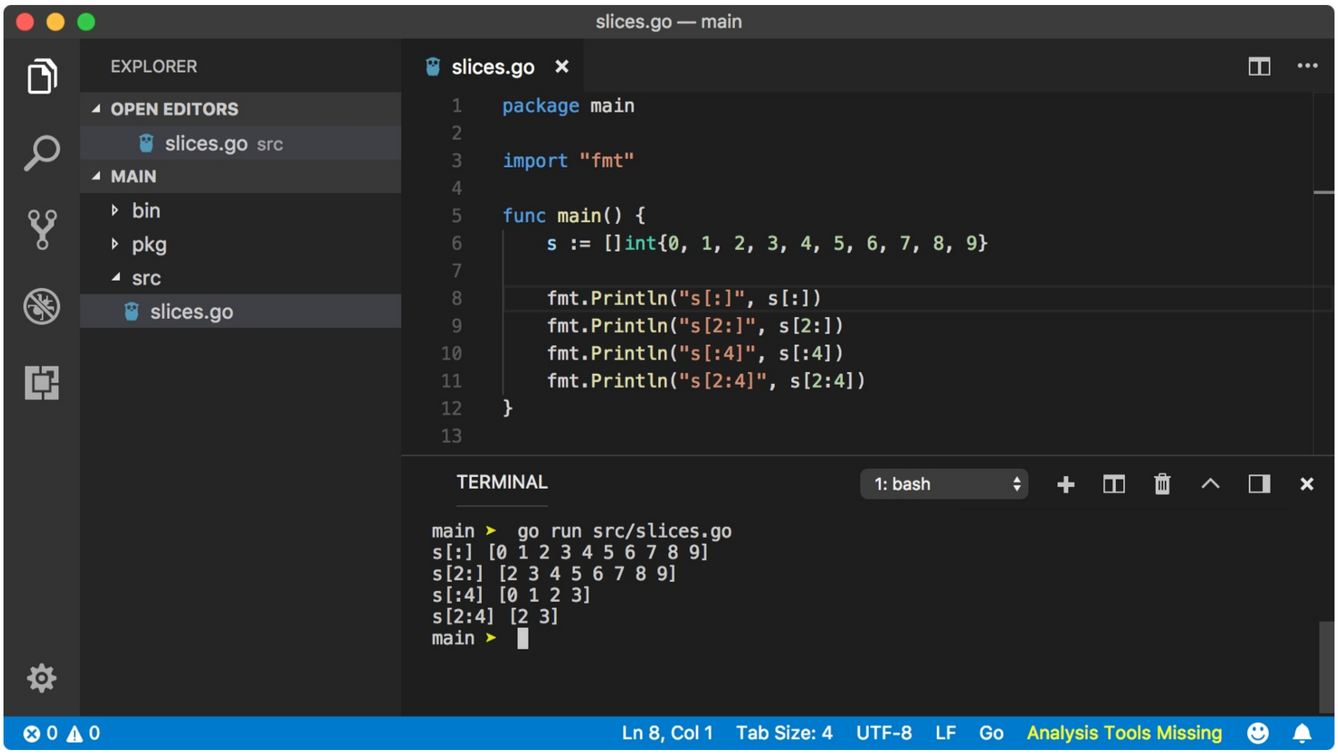
```go
package main

import "fmt"

func main() {
    s1 := make([]int, 0, 10)
    fmt.Println("before -> s1=", s1)

    s2 := []int{1, 2, 3}
    s1 = append(s1, s2...)
    fmt.Println("after -> s1=", s1)
}
```

```
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 a initial index of slice while `end` is last index up to which elements should be extracted hence `end` index is not included. **This syntax returns new slice.**



```go
package main

import "fmt"

func main() {
    s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

    fmt.Println("s[:]", s[:])
    fmt.Println("s[2:]", s[2:])
    fmt.Println("s[:4]", s[:4])
    fmt.Println("s[2:4]", s[2:4])
}
```

```
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 ➤
```

https://play.golang.org/p/lNhNx5KGVrR

In above example, we have 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 0th index till 4th 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 4th index but not including index 4. Hence returns* `[2 3]`

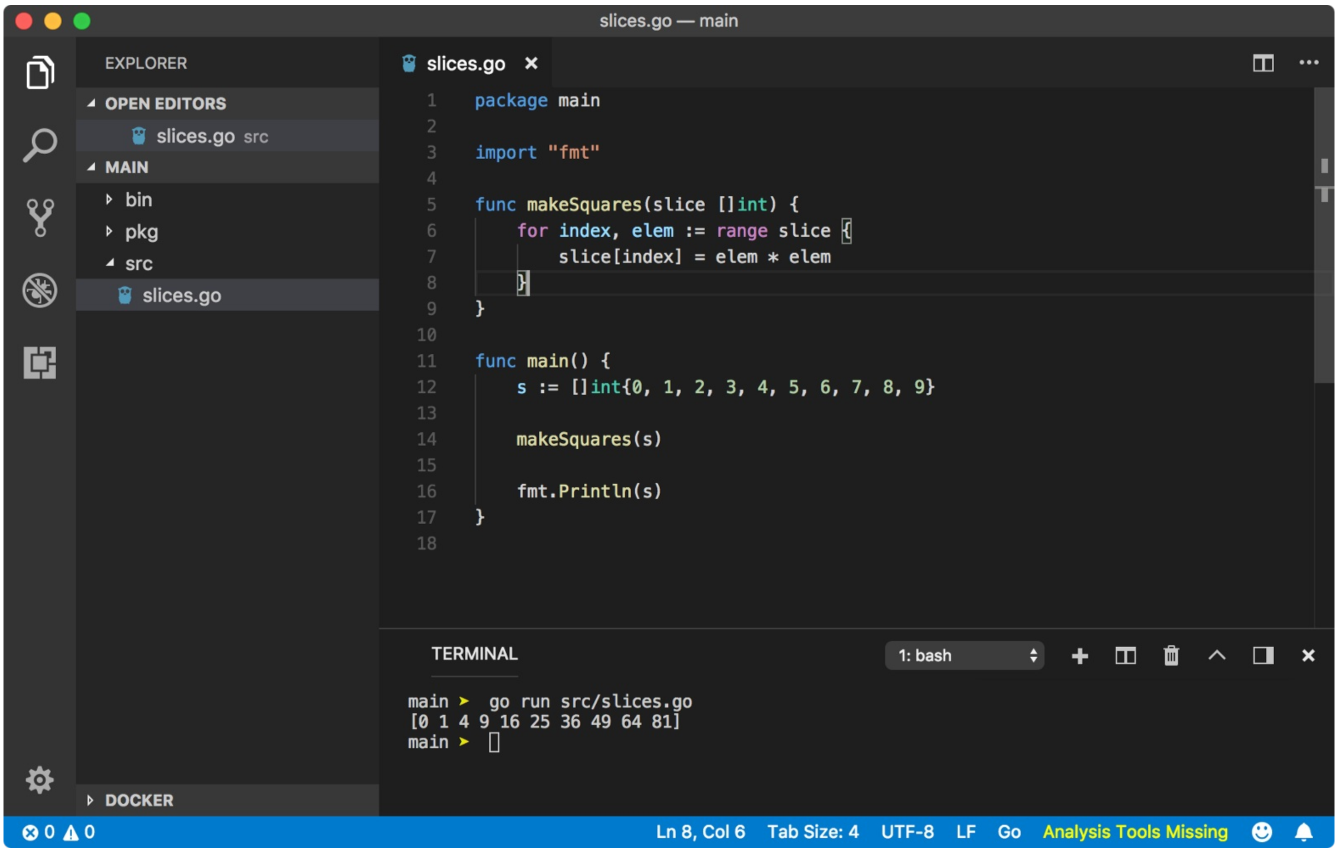
Important thing to remember is that, any slice created by extract operator still references the same underneath array. You can use `copy`, `make` or `append` functions in conjugation to avoid this.

☛ 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 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 references the same underneath array, it looks like that they are passed by reference.



```go
package main

import "fmt"

func makeSquares(slice []int) {
    for index, elem := range slice {
        slice[index] = elem * elem
    }
}

func main() {
    s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

    makeSquares(s)

    fmt.Println(s)
}
```

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

https://play.golang.org/p/p6O0Uqeww1g

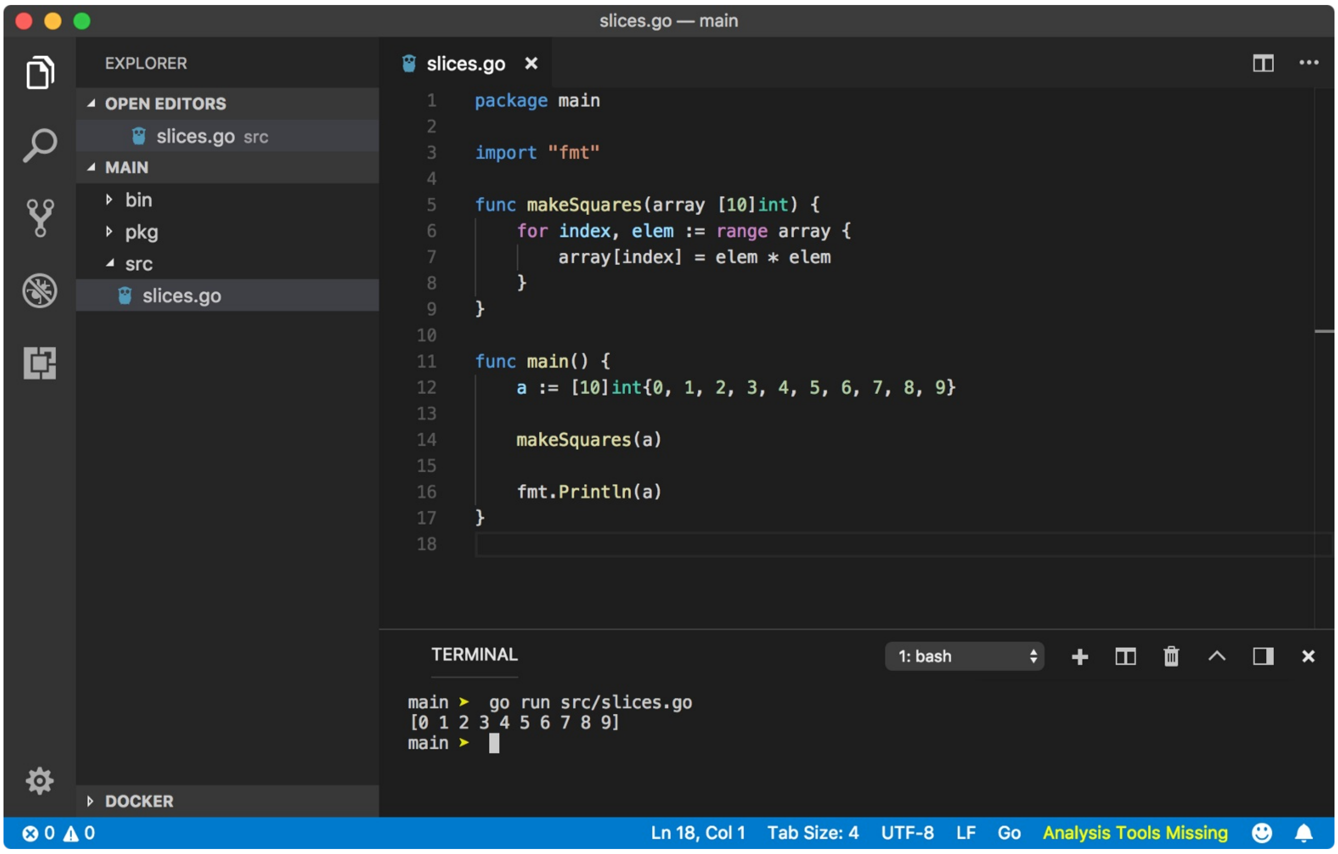
In above example, we have defined `makeSquares` which takes a slice and replaces elements of input slice with their squares. This will yield following result

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

This proves that even though `slice` is passed by value, since it references same underneath `array`, and we can change 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 above program with `array` as input parameter to the function.



```go
package main

import "fmt"

func makeSquares(array [10]int) {
    for index, elem := range array {
        array[index] = elem * elem
    }
}

func main() {
    a := [10]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

    makeSquares(a)

    fmt.Println(a)
}
```

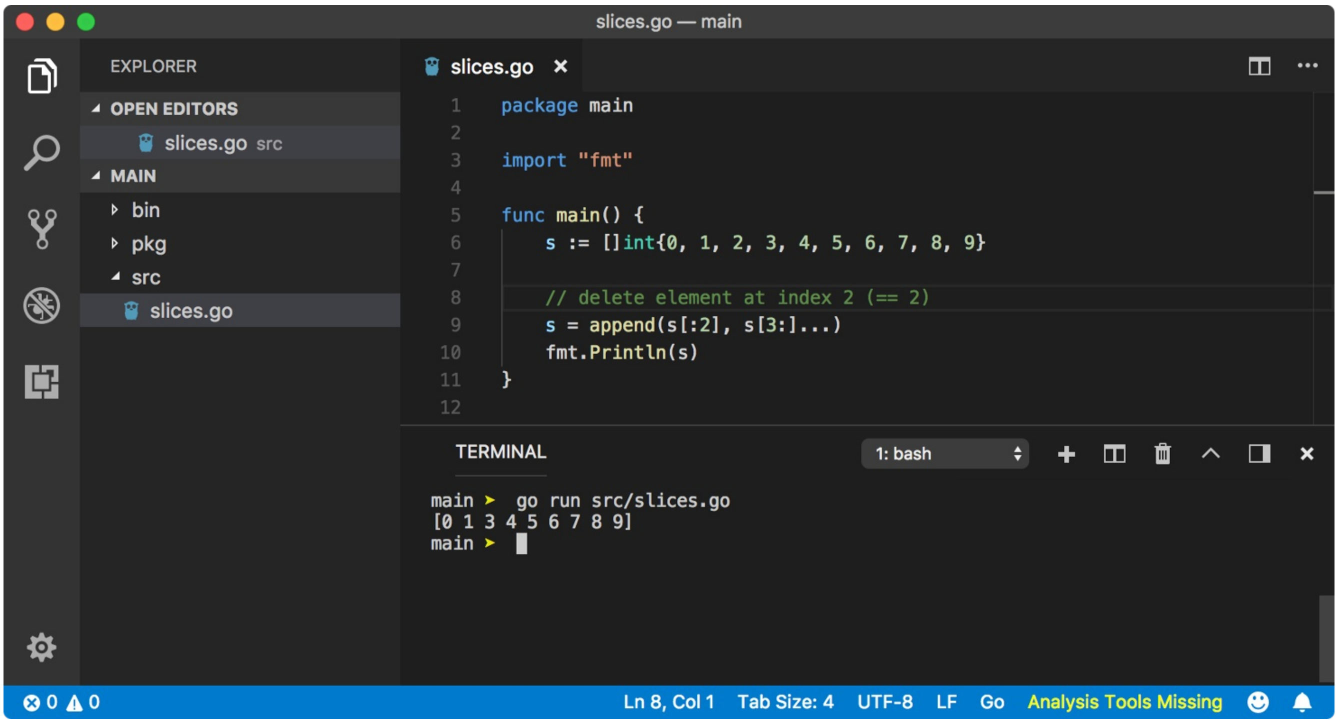
```
main ➤  go run src/slices.go
[0 1 2 3 4 5 6 7 8 9]
main ➤
```

https://play.golang.org/p/qE8grYQ8Q0s

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.



```go
package main

import "fmt"

func main() {
    s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

    // delete element at index 2 (== 2)
    s = append(s[:2], s[3:]...)
    fmt.Println(s)
}
```

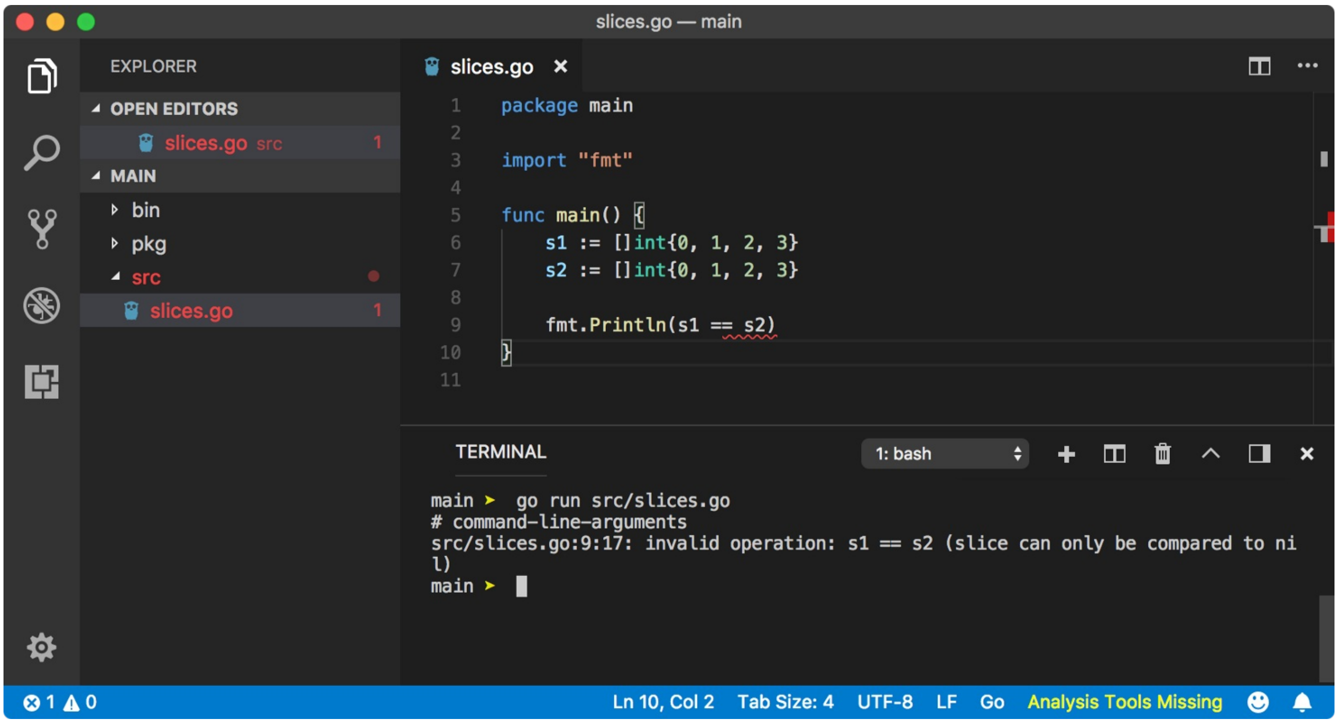
```
main ➤  go run src/slices.go
[0 1 3 4 5 6 7 8 9]
main ➤
```

https://play.golang.org/p/LfLGN2m-uSm

In 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 new slice without index `2`. 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 following program



```go
package main

import "fmt"

func main() {
    s1 := []int{0, 1, 2, 3}
    s2 := []int{0, 1, 2, 3}

    fmt.Println(s1 == s2)
}
```

```
main ➤  go run src/slices.go
# command-line-arguments
src/slices.go:9:17: invalid operation: s1 == s2 (slice can only be compared to ni
l)
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 condition of `nil` or not. If you really need to compare two slice, use `for range` loop to match each elements of the two slices.

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, slices 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 large part of system memory.

Below is a bad program

```
package main

import "fmt"

func getCountries() []string {
 countries := []string{"United states", "United kingdom", "Austrilia", "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, capacity of the countries is `9` means underneath array is holds `9` element (*we know in this case*).

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

```
package main

import "fmt"

func getCountries() (c []string) {
 countries := []string{"United states", "United kingdom", "Austrilia", "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