The anatomy of Arrays in Go

An array is a container that holds data of same type. Arrays in Go have fixed length and they can't be expanded to fit more data.

Uday Hiwarale Follow
Sep 3, 2018 · 7 min read

☛ What is an array

An array is a **collection of the same data type**. For example, an array of integers or an array of strings. Since Go is statically typed language, **mixing different values belonging to different data types in an array is not allowed**.

In Go, an **array has a fixed length**. Once defined with a particular size, the size of an array cannot be increased or decreased. But that problem can be solved using **slices**which we will learn in the next lesson.

> *Array is a composite data type because it is composed of primitive data types like* `int`, `string`, `bool` *etc.*

☛ How to declare an array?

An array is a type in itself. This type is defined like `[n]T` where `n` is the number of elements an array can hold and `T` is a data type like `int` or `string`.

Hence to define a variable which is an array of `3` elements of the type `int`, it has the following syntax

```
package main

import "fmt"

func main() {
        var a [3]int

fmt.Println(a)
}
```

In the above program, `a` is an array of **3 integer elements** but we haven't assigned any values to individual array elements. What is your guess of `Println` statement. **What is the value of an empty array?**

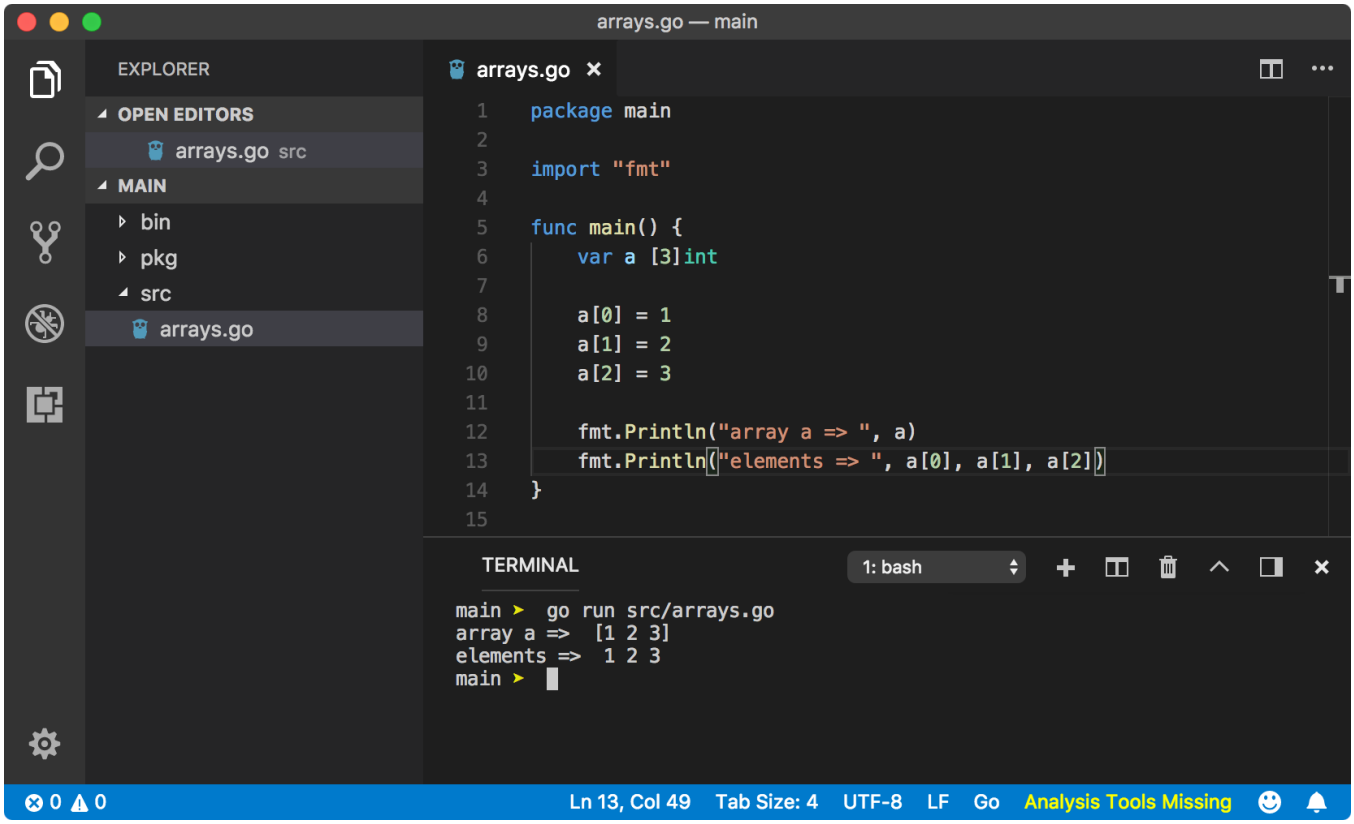
Since we haven't assigned any value to `a`, we just defined the array but not the value of array elements. **So it will have zero value of its data type**. For `int`, its zero value if `0` hence `Println` statement will print an array of 3 zeros.

```
[0 0 0]
```

☛ Value assignment

We can individually assign values to each element of an array using their position in the array AKA `index` number. In Go, the array index starts from `0` which is the first element, hence last element index will be `n-1` where `n` is the length of the array.

To access any element in the array, we need to use `a[index]` syntax where `a` is an array variable. So let's take our earlier example and modify it a little.



https://play.golang.org/p/_4DMJ3iRqRF

In the above program, we assigned new values to all 3 elements in the array using `a[index]` syntax.

☛ Initial value

It would be pretty difficult to assign value to each element of an array if the array is big. Hence Go provides short-hand syntax to define an array with an initial value or array elements with pre-defined values. The syntax to define an array with initial values is as following

```
var a [n]T = [n]T{V1,V2,...,Vn}
```

In the previous example, if values of array elements would be `1, 2, 3` then syntax to define an array would be as following

```
var a [3]int = [3]int{1, 2, 3}
```

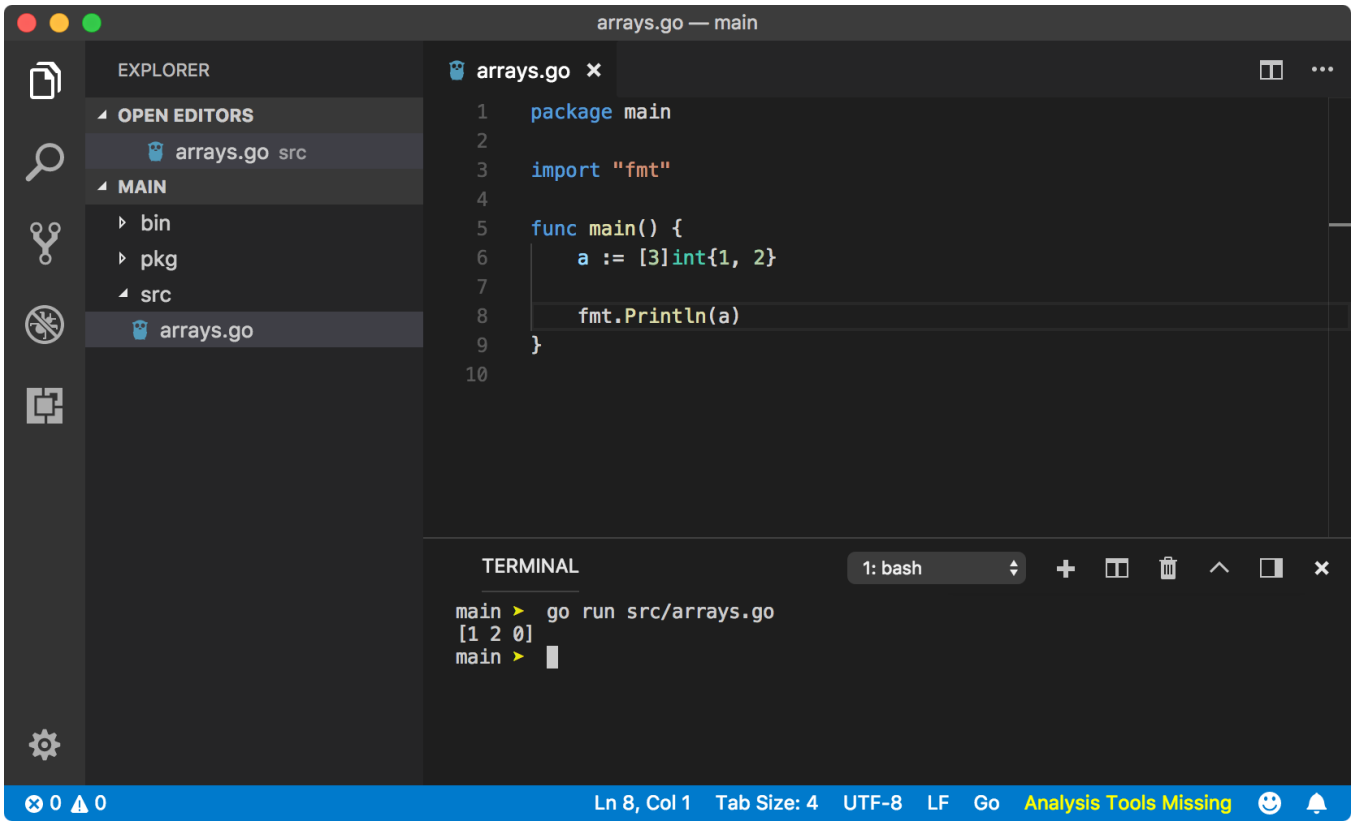
You can also drop type from the left-hand statement and Go will infer the type from the array definition.

```
var a = [3]int{1, 2, 3}
```

Or you could use shorthand syntax `:=`, dropping `var`

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

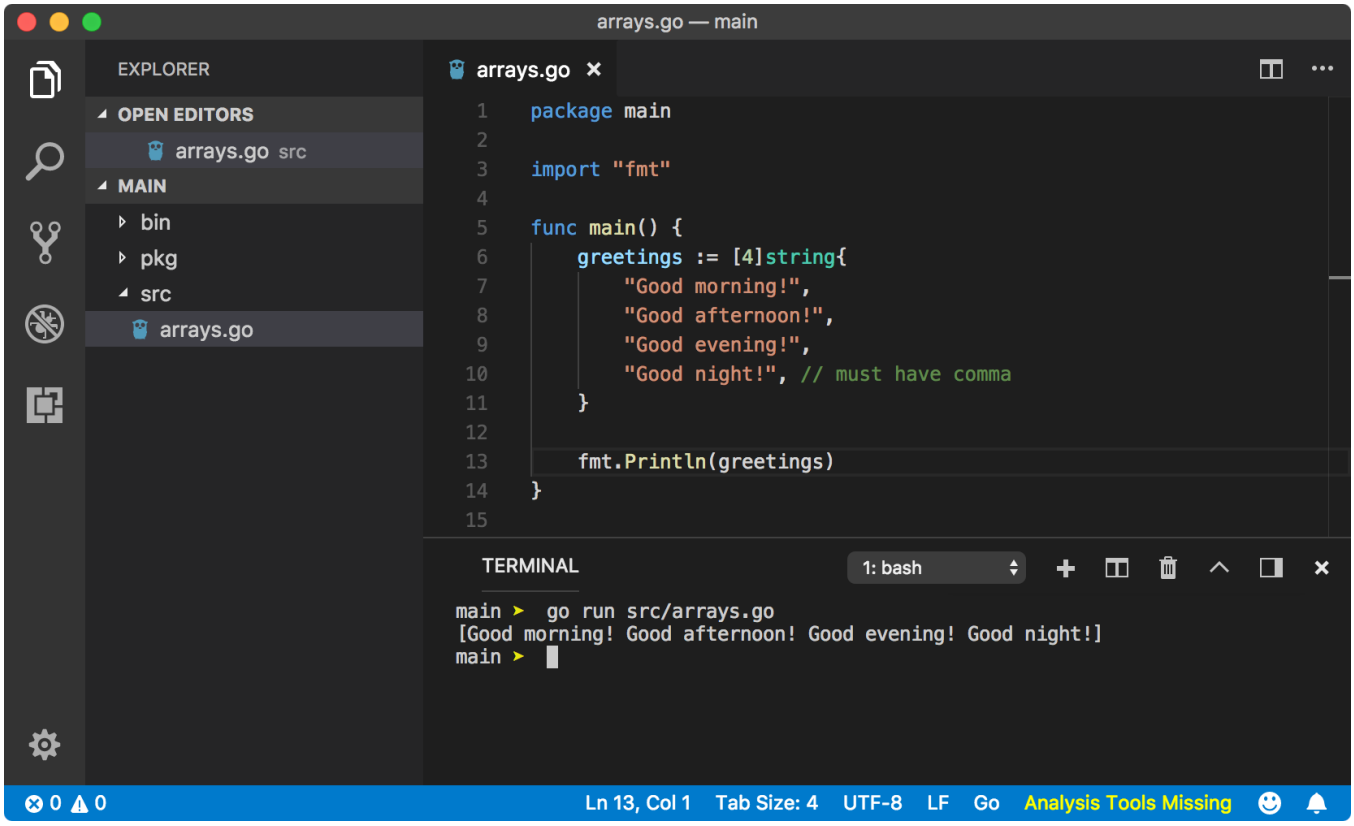
There is absolutely no need to define all elements of an array. In the above example, we could have defined the first two elements, leaving the third element a zero-value.



https://play.golang.org/p/J4x7X8Vg73G

☛ Multi-line initial value

You can define an array with initial values over multiple lines but since these values are comma-separated, you need to make sure to add a comma at the end of the last element. Why? Read further.

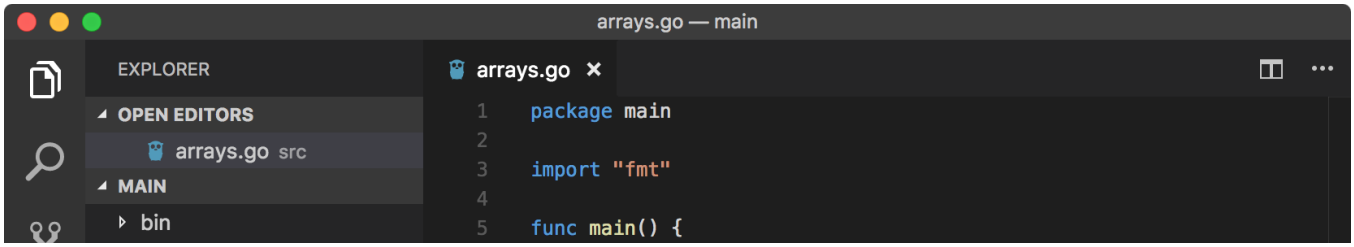


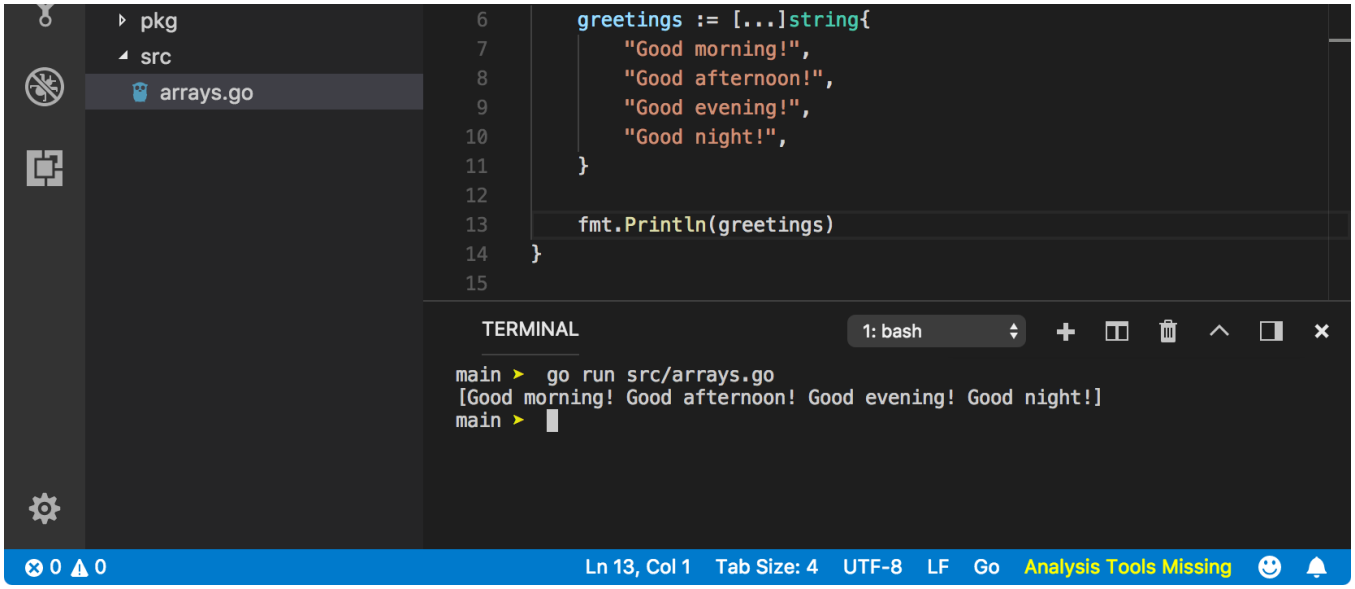
https://play.golang.org/p/IcwpC-fQQBj

Look carefully, we have used comma (`,`) at the end of last element of the array. This comma is necessary as if it wouldn't be there, Go would have added a semicolon (`;`) by the rule which would have crashed the program.

☛ Automatic array length declaration

Sometimes, we don't know the length of an array while typing its elements. Hence Go provide `...` operator to put in place of `n` in `[n]T` array type syntax. Go compiler will find the length on its own. You can only use this operator when you are defining an array with an initial value.



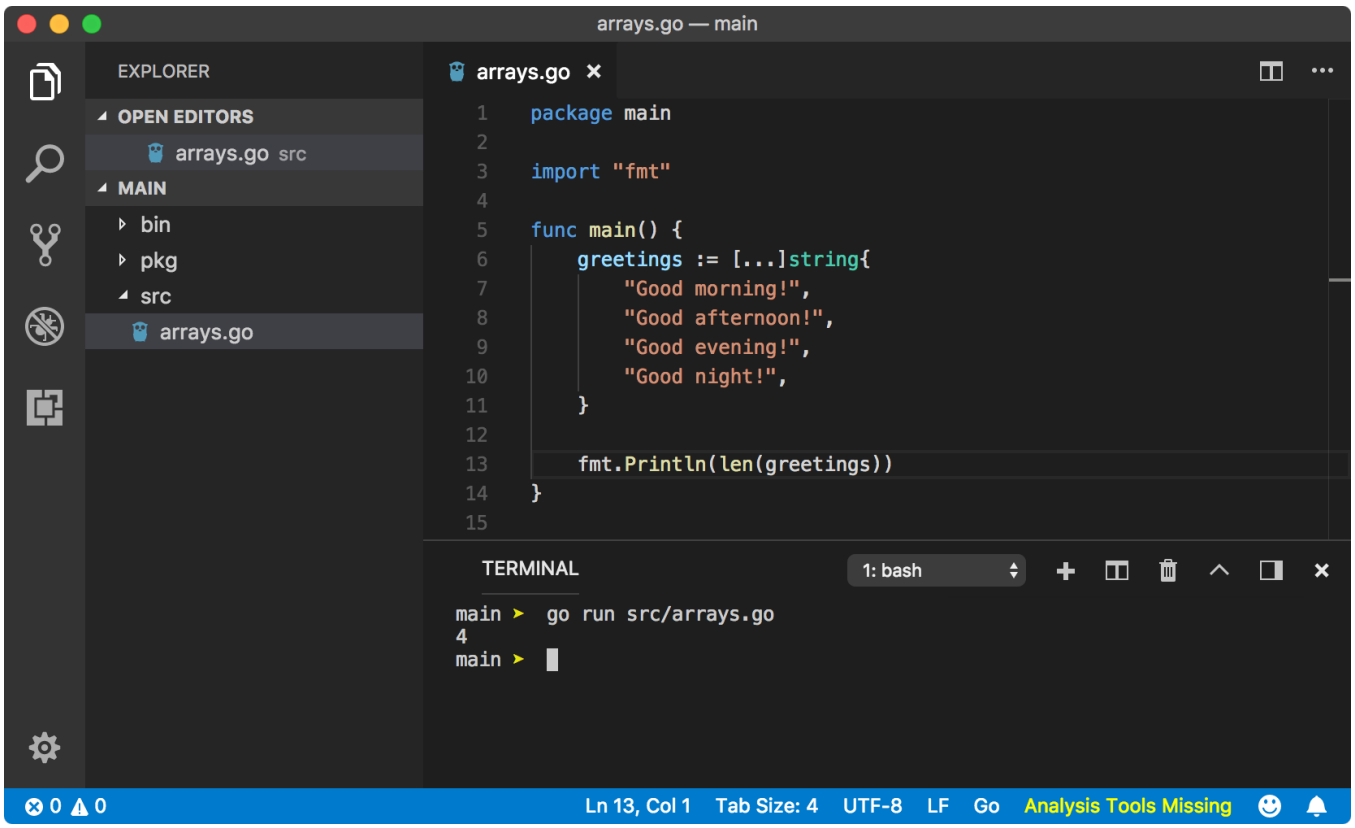


https://play.golang.org/p/UKl32kgngnE

The above program will print the same result because Go compiler guesses the value of `4` from the number of elements of the array which are `4`.

☛ Find the length of an array

Go provide a **built-in** function `len` which is used to calculate the length of many data types, here, in this case, we can use it to calculate the length of an array.



https://play.golang.org/p/wV9edsKhuh4

☛ Array comparison

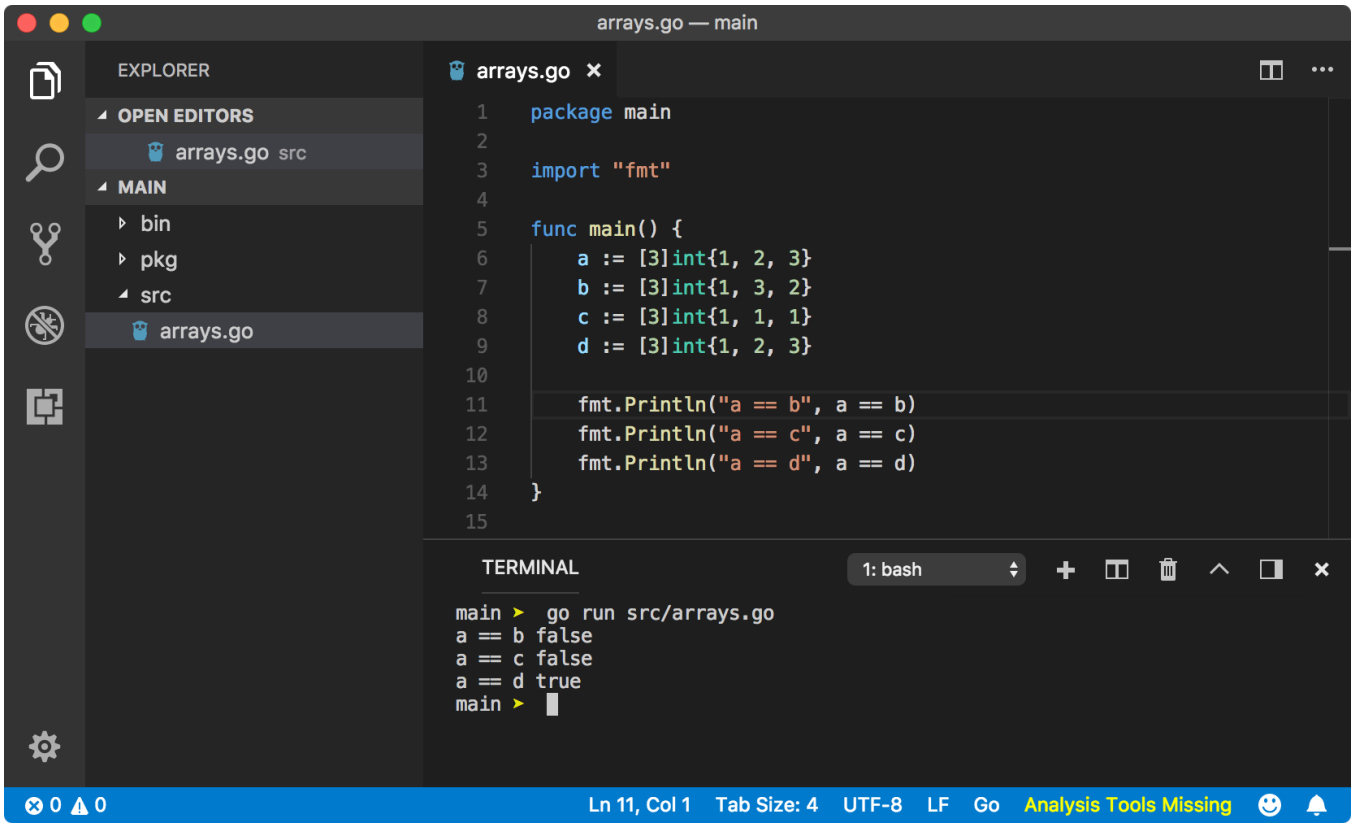
As we discussed earlier in array definition, the array is a type in itself. `[3]int` is different from `[4]int` which is very different from `[4]string`. It's like comparing `int == string` or `apple == orange` which is invalid and nonsense. Hence these arrays cannot be compared with each other, unlike other programming languages.

While `[3]int` can be compared with `[3]int` even if their array elements do not match, because they have the same data type.

For an array to be the equal or the same as the second array, both array should be of the same type, must have the same elements in them and all elements must be in the same order. In that case, `==` comparison will be `true`. If one or more of these conditions do not match, it will return `false`.

Go matches first the data type and then secondly each element of the array with an element of another array by the index.

Let's have a look at below program.



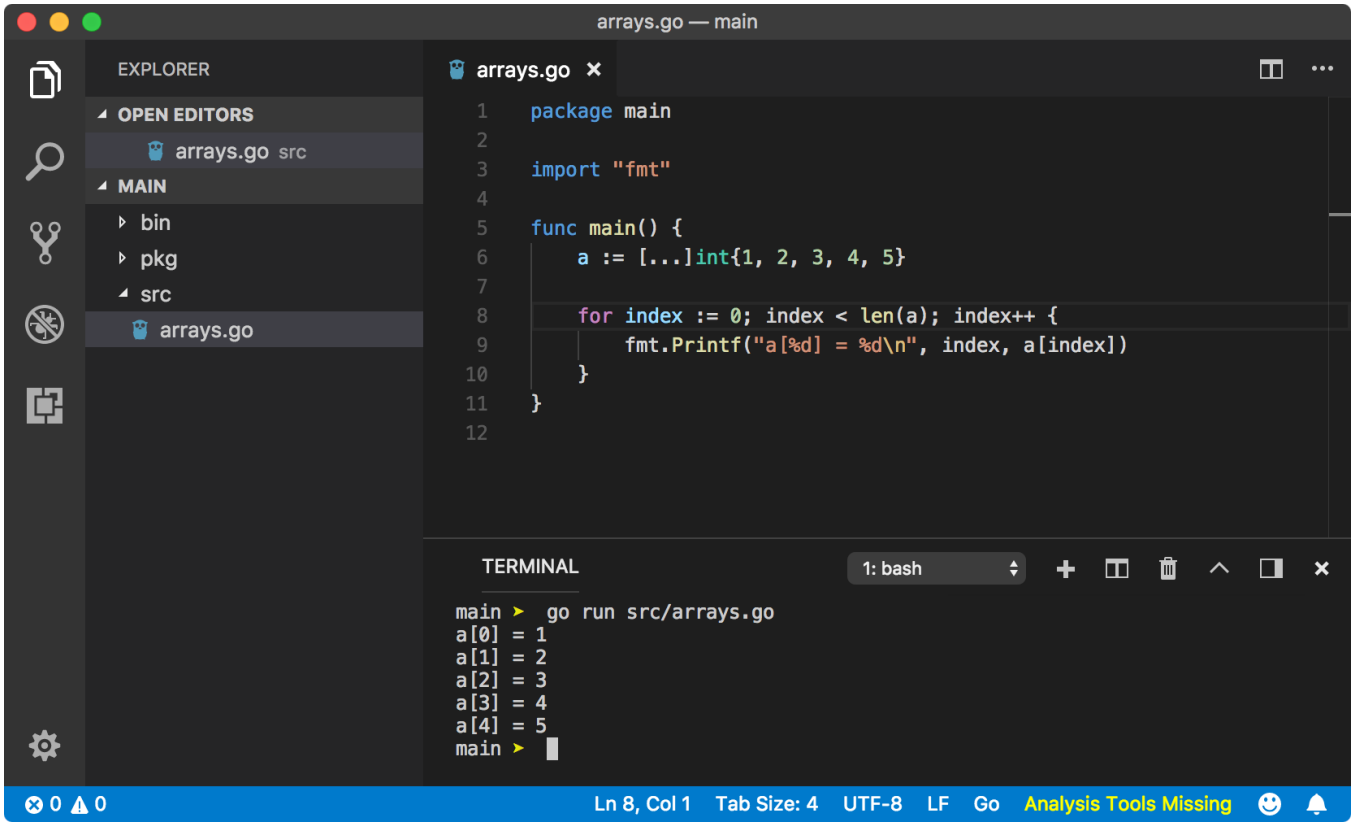
https://play.golang.org/p/U933frU9GqI

So `a`, `b`, `c` and `d`, all of them have the same data type of `[3]int`. In the first comparison, `a == b`, since `a` and `b` both contain the same element but different order, this condition will be `false`. In the second comparison `a == c`, since `c` contains completely different elements than `a`, this condition will be `false`.

But in the case of the third comparison, since both `a` and `d` contains the same elements with the same order, `a == d` condition will be `true`. Hence above program prints below result.

☛ Array iteration

To iterate over an array, we can use `for` loop.

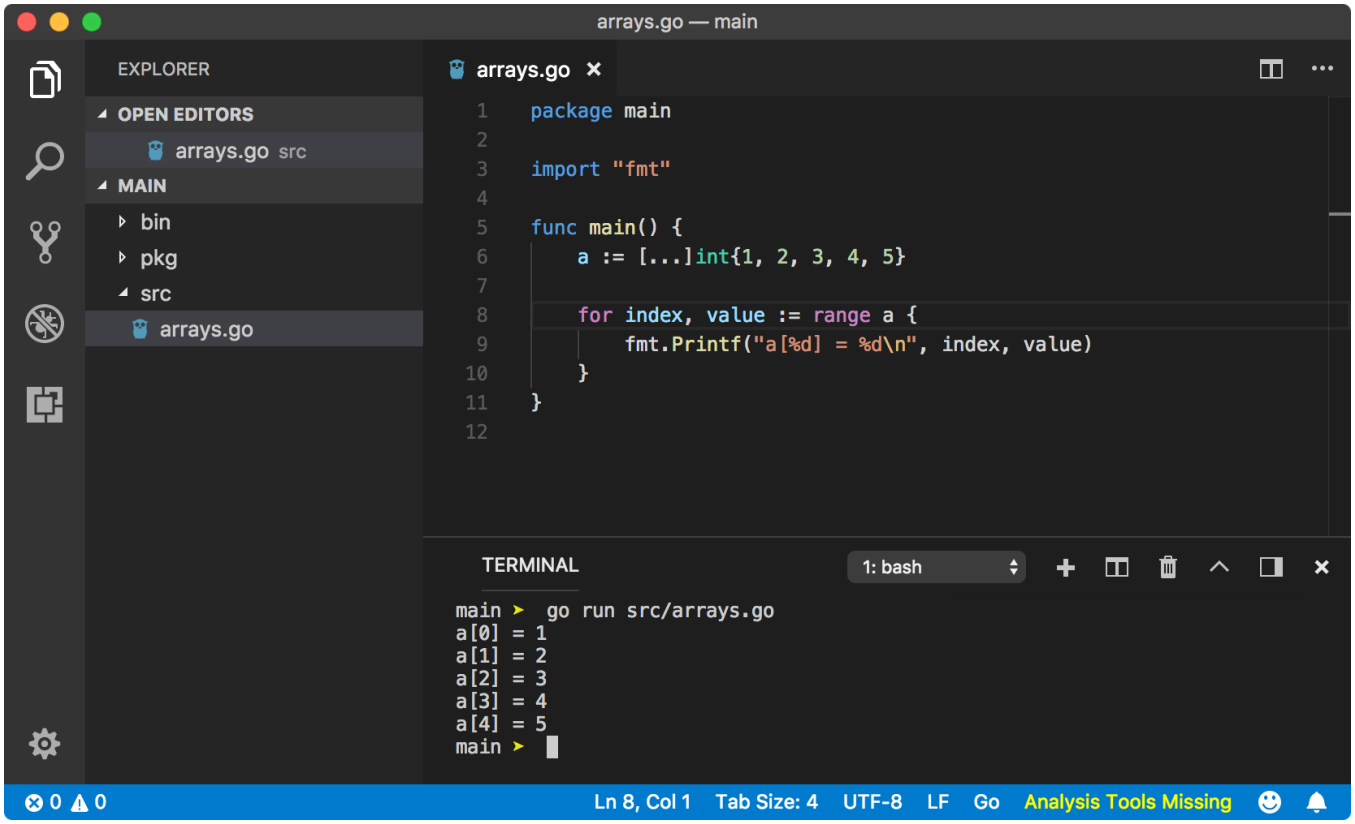


https://play.golang.org/p/v5oGN2qQbgN

In the above example, since the element `index` in the array is always less than `len(a)`, we can print each element of an array using simple for loop.

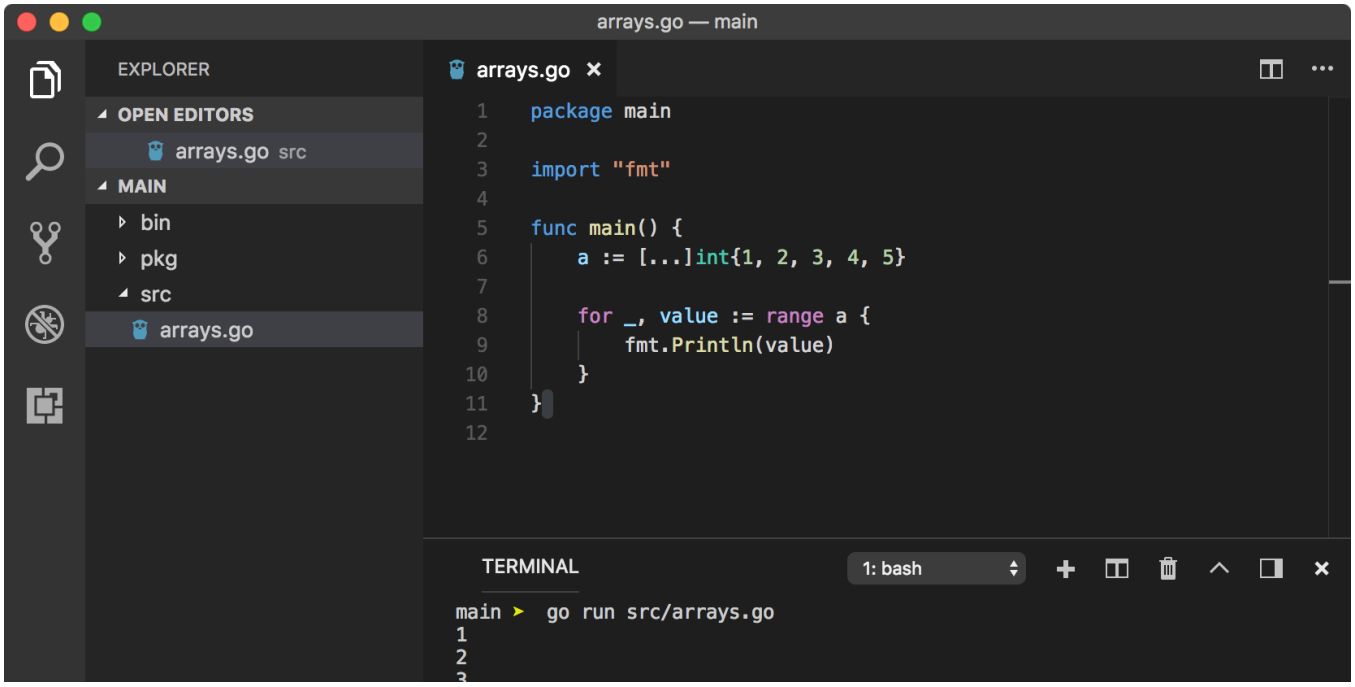
But I absolutely hate above way of iterating an array where I need to use `index` to get the element of an array. But no worries, Go provides `range` operator which

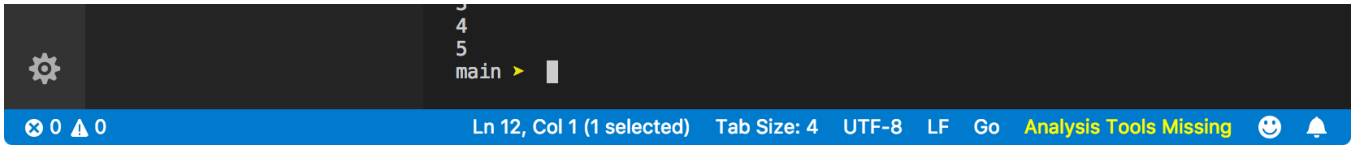
returns `index` and `value` of each element of an array in `for` loop.



https://play.golang.org/p/XY9pONQaYng

`range` operator returns `index` and `value` of the element associated with an element until all elements in the array are finished. If you are not interested in the **index**, we can just assign it to the **blank** identifier.



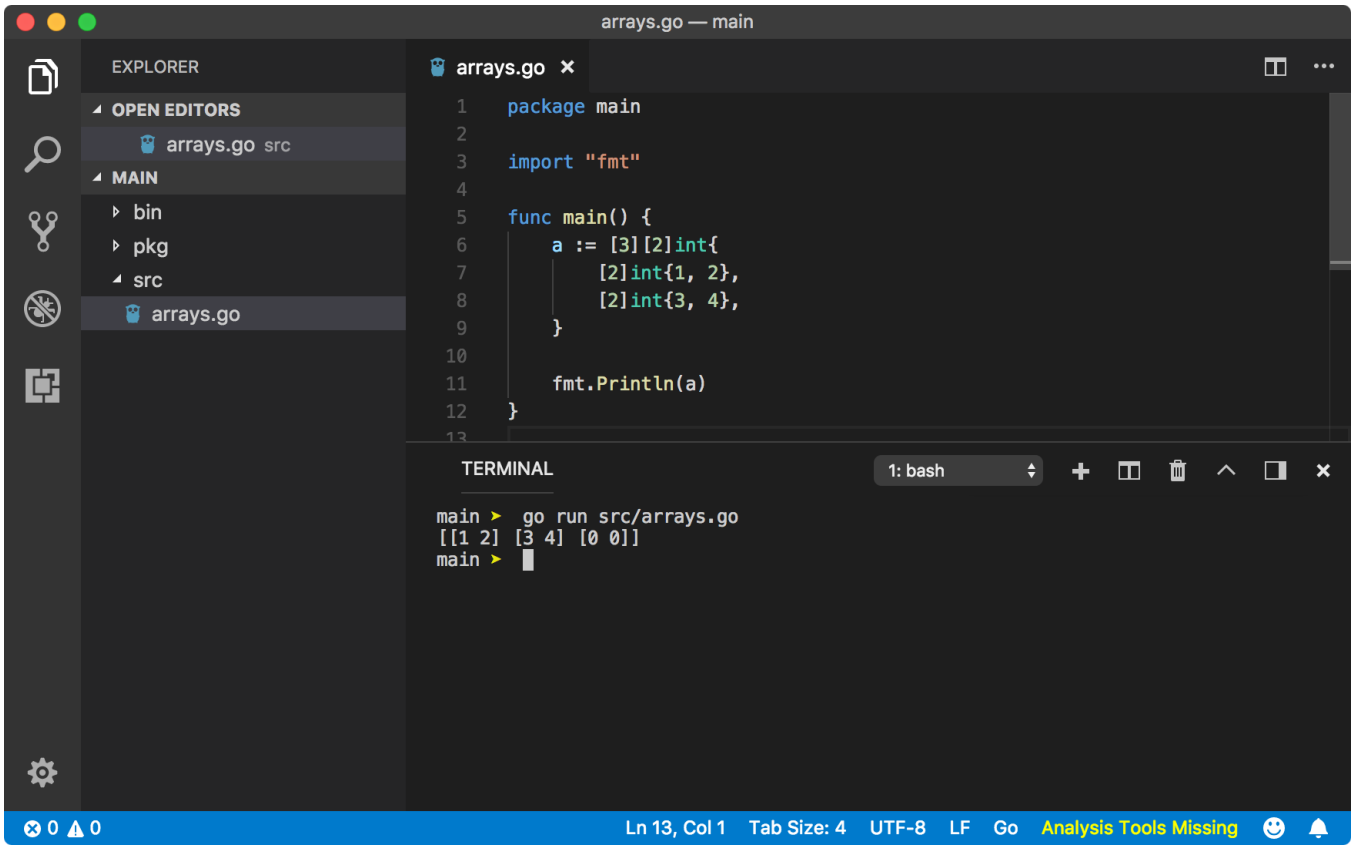


https://play.golang.org/p/z0_v0CKquXU

☛ Multi-dimensional arrays

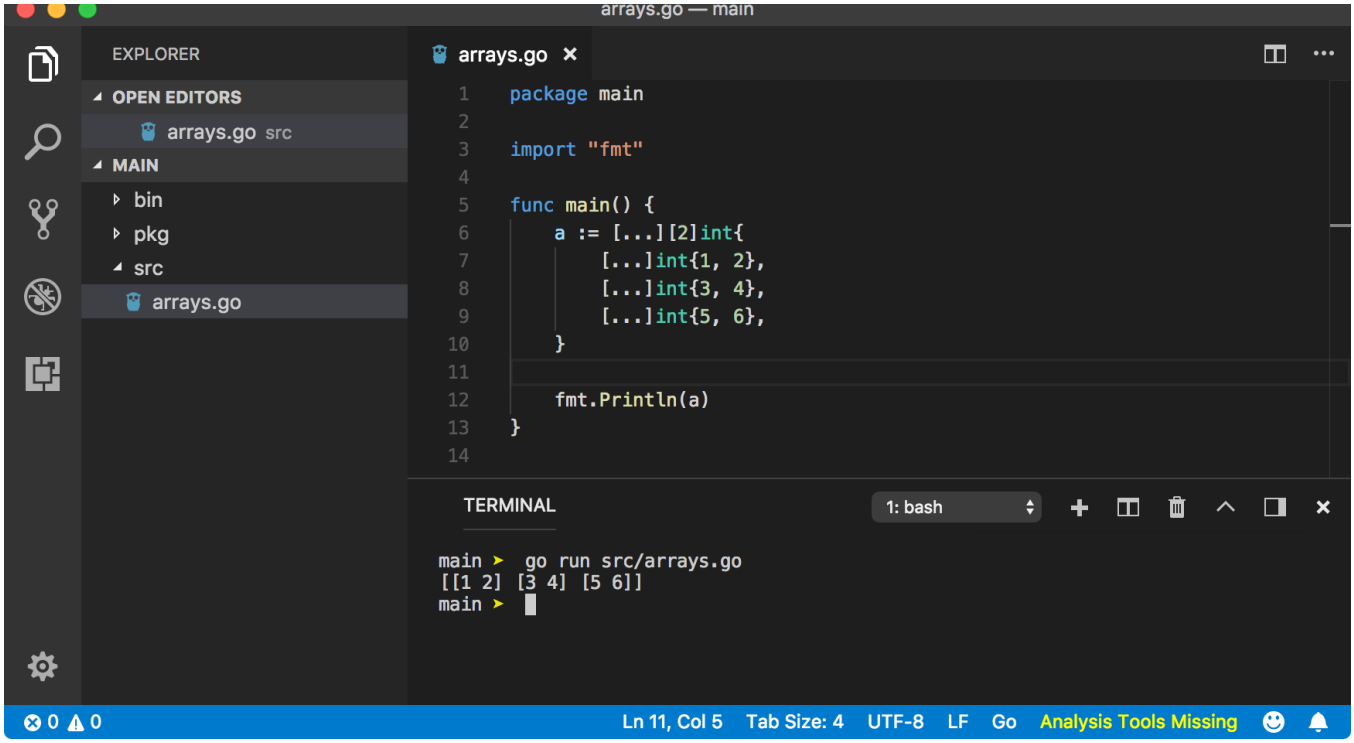
When array elements of an array are arrays, then it's called a **multi-dimensional array**. As from the definition of the array, an **array is a collection of same data types** and **array is a type in itself**, **a multi-dimensional array must have arrays that belong to the same data type**.

Syntax to write multi-dimensional array is `[n][m]T` where `n` is the number of elements in the array and `m` is the number of elements in inner array. So technically, we can say array contains `n` elements of type `[m]T`.



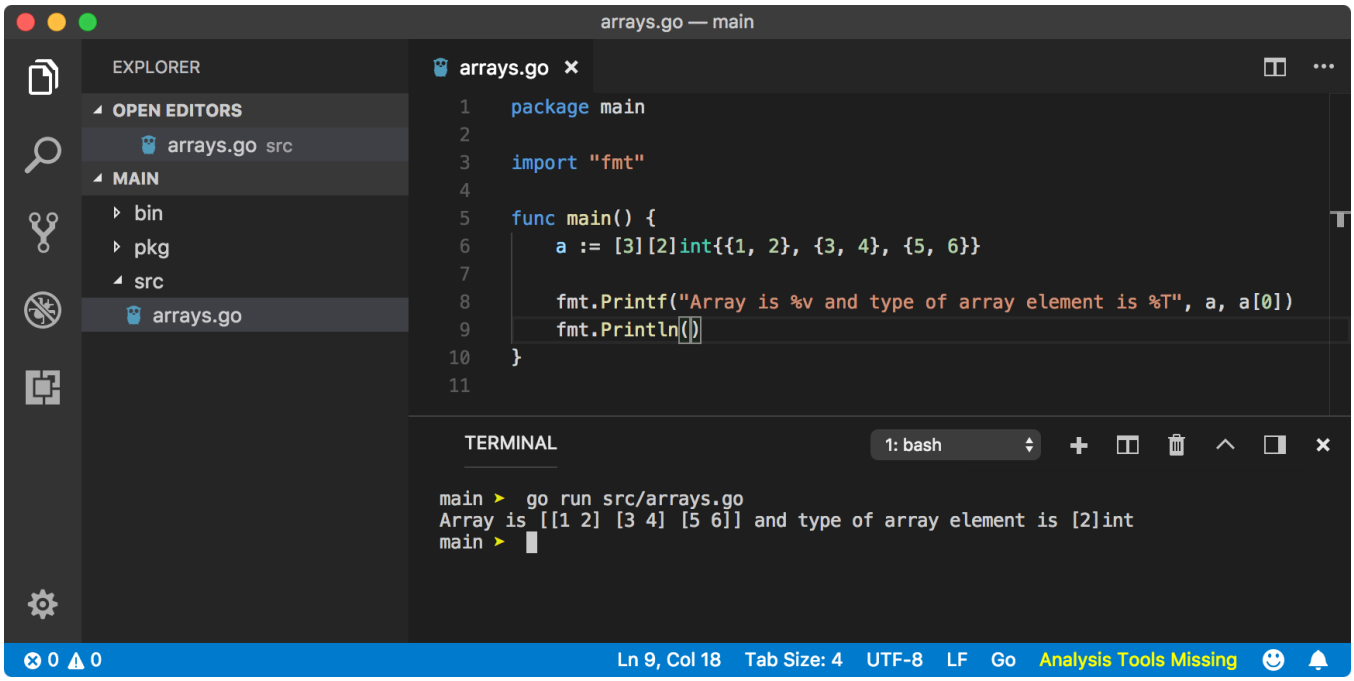
https://play.golang.org/p/7Hfeb30HD-H

Since we can use `...` operator to guess the size of an array, Above program, can also be written as



https://play.golang.org/p/w7oyAb_Acik

But Go provides an **amazing short syntax** to write multi-dimensional array.



https://play.golang.org/p/_ZMipOrWy1R

In the above program, Go already knows the type of array elements which is `[2]int` hence no need to mention it again. You could also use `...` operator like

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

Iterating over a multi-dimensional array is no different than that of a simple array. Only in case of a multi-dimensional array, the value of the array element is also an array which needs to be iterated over again. Hence, you will see `for` loop nested under another `for` loop like below.

```
for _, child := range parent {
        for _, elem := range child {
                ...
        }
}
```

Passed by value

When you pass an array to a function, they are passed by `value` like `int` or `string` data type. The function receives only a copy of it. Hence, when you make changes to an array inside a function, it won't be reflected in the original array.

Why I needed to mention that, because it's not the case with `slice` which you will see in upcoming tutorials.