#### Slices in Go

This section explains the concept of slices in Go and its details such as slicing, appending slices, nil slices and finding slice length

#### WE'LL COVER THE FOLLOWING ^

- Slices
- Slicing a slice
  - Syntax
  - Making slices
  - Appending to a slice
- Length
  - Nil slices
- Resources

## Slices #

Slices wrap arrays to give a more general, powerful, and convenient interface to sequences of data. Except for items with an explicit dimension such as *transformation matrices*, most array programming in Go is done with slices rather than simple arrays.

Slices hold references to an underlying array, and if you assign one slice to another, both refer to the same array. If a function takes a slice argument, changes it makes to the elements of the slice will be visible to the caller, analogous to passing a pointer to the underlying array.

A slice points to an array of values and also includes a length. Slices can be resized since they are just a wrapper on top of another data structure.

[]T is a slice with elements of type T.

```
Environment Variables

Key: Value:

GOPATH /go

package main

import "fmt"

func main() {
    p := []int{2, 3, 5, 7, 11, 13}
    fmt.Println(p)
    // [2 3 5 7 11 13]
}
```

# Slicing a slice #

Slices can be re-sliced, creating a new slice value that points to the same array.

### Syntax #

```
s[lo:hi]
```

evaluates to a slice of the elements from 10 through hi-1, inclusive. Thus

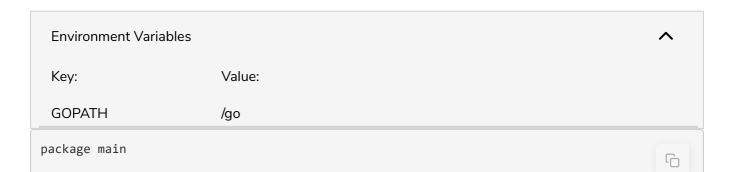
```
s[lo:lo]
```

is empty and

```
s[lo:lo+1]
```

has one element.

Note: 10 and hi would be integers representing indexes.



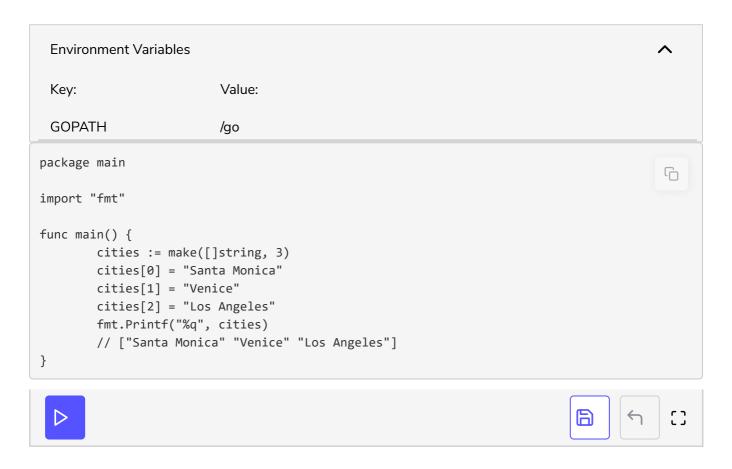
Let's take a look at another example below:



at the index **0** and **1** of the array and **b** containing the elements at index **1** and **2** of the array.

### Making slices #

Besides creating slices by passing the values right away (slice literal), you can also use make. You create an empty slice of a specific length and then populate each entry:



It works by allocating a zeroed array and returning a slice that refers to that array.

#### Appending to a slice #

Note however, that you would get a runtime error if you were to do that:

```
cities := []string{}
cities[0] = "Santa Monica"
```

As explained above, a slice is seating on top of an array, in this case, the array is empty and the slice can't set a value in the referred array. There is a way to do that though, and that is by using the append function:

```
package main
import "fmt"
```

```
func main() {
    cities := []string{}
    cities = append(cities, "San Diego")
    fmt.Println(cities)
    // [San Diego]
}
```

You can append more than one entry to a slice:

```
package main

import "fmt"

func main() {
    cities := []string{}
    cities = append(cities, "San Diego", "Mountain View")
    fmt.Printf("%q", cities)
    // ["San Diego" "Mountain View"]
}
```

And you can also append a slice to another using an ellipsis:

```
package main
import "fmt"

func main() {
    cities := []string{"San Diego", "Mountain View"}
    otherCities := []string{"Santa Monica", "Venice"}
    cities = append(cities, otherCities...)
    fmt.Printf("%q", cities)
    // ["San Diego" "Mountain View" "Santa Monica" "Venice"]
}
```

Note that the ellipsis is a built-in feature of the language that means that the element is a collection. We can't append an element of type slice of strings ([]string) to a slice of strings, only strings can be appended. However, using the ellipsis (...) after our slice, we indicate that we want to append each element of our slice. Because we are appending strings from another slice, the

compiler will accept the operation since the types are matching.

You obviously can't append a slice of type []int to another slice of type []string.

## Length #

At any time, you can check the length of a slice by using len:

```
package main
                                                                                         6
import "fmt"
func main() {
        cities := []string{
                "Santa Monica",
                "San Diego",
                "San Francisco",
        }
        fmt.Println(len(cities))
        countries := make([]string, 42)
        fmt.Println(len(countries))
        // 42
}
                                                                                          []
```

#### Nil slices #

The zero value of a slice is nil. A nil slice has a length and capacity of  ${\bf 0}$ .

```
package main

import "fmt"

func main() {
    var z []int
    fmt.Println(z, len(z), cap(z))
    // [] 0 0
    if z == nil {
        fmt.Println("nil!")
    }
    // nil!
}
```

#### Resources +

For more details about slices read the links below:

- Go slices, usage and internals
- Effective Go slices
- Append function documentation
- Slice tricks
- Effective Go slices
- Effective Go two-dimensional slices
- Go by example slices

Next lesson discusses *range* form of for loops for iterating over slices. Read on to find out more!