

# Table of Contents

Solutions.....	1
1    Code 1.....	1
1.1 <i>Running the code</i> .....	1
1.1.1 <i>Existing condition</i> .....	1
1.1.2 <i>Updated Condition</i> .....	1
1.2 <i>Explanation</i> .....	3
2    Code 2.....	4
2.1 <i>Running the code</i> .....	4
2.1.1 <i>Existing Condition</i> .....	4
2.1.2 <i>Updated Condition</i> .....	5
2.2 <i>Explanation</i> .....	6

## List of Code Snippets

Code 1: The given main source code for first problem.	1
Code 2: Updated Code Block	2
Code 3: <code>launch.json</code> file that assists the successful running of Program 1.	2
Code 4: Go module initialization and running.	2
Code 5: The given main source code for the second program.	4
Code 6: Code snippet for the updated code from the original code.	5
Code 7: Original non edited portion of code.	6
Code 8: Updated edited portion of code.	6

## List of Figures

Figure 1: Error Message after running Code 1 in as is condition.	1
Figure 2: Running output from program 1 after corrections.	3
Figure 3: Initial unaltered program output.	5
Figure 4: Correct and updated results from update to the code.	6

## Solutions

### 1 Code 1

```
package main

import (
    "fmt"
    "math/rand"
)

func appendRand(a []float64) {
    a = append(a, rand.Float64())
}

func main() {
    a := make([]float64, 0)
    for i := 0; i < 10; i++ {
        appendRand(a)
        fmt.Printf("%d: %v\n", i, a)
    }
}
```

Code 1: The given main source code for first problem.

#### 1.1 Running the code

##### 1.1.1 Existing condition

When the code is run in its given as is condition, the following Figure 1 portrays the error message that is displayed.

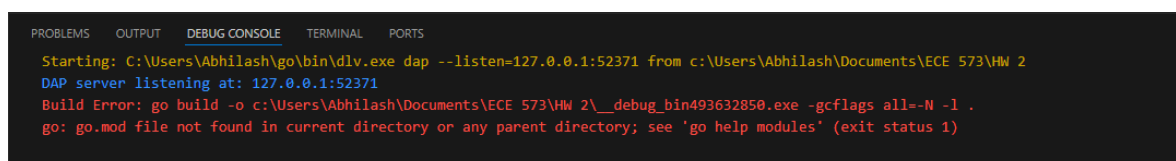


Figure 1: Error Message after running Code 1 in as is condition.

##### 1.1.2 Updated Condition

The following is the code block that is updated from the original code.

```

package main

import (
    "fmt"
    "math/rand"
)

func main() {
    a := make([]float64, 0)
    for i := 0; i < 10; i++ {
        a = appendRand(a)
        fmt.Printf("%d: %v\n", i, a)
    }
}

func appendRand(a []float64) []float64 {
    return append(a, rand.Float64())
}

```

Code 2: Updated Code Block

Along with this, a `launch.json` file is also created and the Code 3 represents the same.

```

{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Launch Package",
            "type": "go",
            "request": "launch",
            "mode": "auto",
            "program": "${fileDirname}"
        }
    ]
}

```

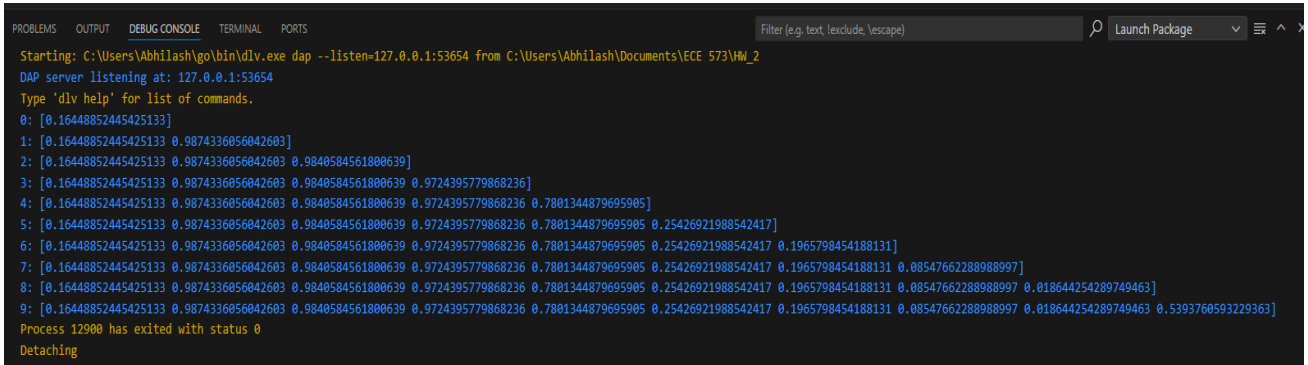
Code 3: `launch.json` file that assists the successful running of Program 1.

```
>> go mod hw2
```

```
>> go mod tidy
```

Code 4: Go module initialization and running.

Additionally, in the working file directory, using the terminal, go module is initialized and run using the above Code 4. Thus, the output for the same program is illustrated in the below Figure 2.



```
Starting: C:\Users\Abhilash\go\bin\div.exe dap --listen=127.0.0.1:53654 from C:\Users\Abhilash\Documents\ECE 573\HW_2
DAP server listening at: 127.0.0.1:53654
Type 'div help' for list of commands.
0: [0.16448852445425133]
1: [0.16448852445425133 0.9874336056042603]
2: [0.16448852445425133 0.9874336056042603 0.9840584561800639]
3: [0.16448852445425133 0.9874336056042603 0.9840584561800639 0.9724395779868236]
4: [0.16448852445425133 0.9874336056042603 0.9840584561800639 0.9724395779868236 0.7801344879695905]
5: [0.16448852445425133 0.9874336056042603 0.9840584561800639 0.9724395779868236 0.7801344879695905 0.25426921988542417]
6: [0.16448852445425133 0.9874336056042603 0.9840584561800639 0.9724395779868236 0.7801344879695905 0.25426921988542417 0.1965798454188131]
7: [0.16448852445425133 0.9874336056042603 0.9840584561800639 0.9724395779868236 0.7801344879695905 0.25426921988542417 0.1965798454188131 0.08547662288988997]
8: [0.16448852445425133 0.9874336056042603 0.9840584561800639 0.9724395779868236 0.7801344879695905 0.25426921988542417 0.1965798454188131 0.08547662288988997 0.018644254289749463]
9: [0.16448852445425133 0.9874336056042603 0.9840584561800639 0.9724395779868236 0.7801344879695905 0.25426921988542417 0.1965798454188131 0.08547662288988997 0.018644254289749463 0.5393760593229363]
Process 12900 has exited with status 0
Detaching
```

Figure 2: Running output from program 1 after corrections.

## 1.2 Explanation

The ‘appendRand’ function in this application was the primary source of trouble. It wasn't really altering the original slice supplied to it, even though it seemed to add a random float to it. The way Go handles slices that are provided to functions is the cause of this.

Slices in Go are supplied by value, but this value also contains a reference to the array that is behind. If add is used within ‘appendRand’, a new slice with more capacity is created if necessary. The assignment to an internal ‘appendRand’, however, has no effect on the original slice in the main function because this new slice is local to the function.

Consequently, there would be no problems in the program's operation, but the slice in main would always be empty.

The main function now correctly builds up the slice over the 10 iterations, and each ‘fmt.Printf’ call shows the growing slice.

This method complies with Go's guidelines about the operation of slices and how they need to be managed when adjustments are required across function boundaries. It transforms a function that was effectively a no-op into one that appropriately expands and updates the slice as intended.

## 2 Code 2

```
package main

import (
    "fmt"
)

type Vertex struct {
    X, Y int
}

func (v *Vertex) Move(dx, dy int) {
    v.X += dx
    v.Y += dy
}

func (v Vertex) String() string {
    return fmt.Sprintf("(%d,%d)", v.X, v.Y)
}

func main() {
    a := make([]Vertex, 0)
    for i := 0; i < 10; i++ {
        a = append(a, Vertex{X: i, Y: i * 2})
    }
    fmt.Printf("before move: %v\n", a)
    for _, v := range a {
        v.Move(1, 2)
    }
    fmt.Printf("after move: %v\n", a)
}
```

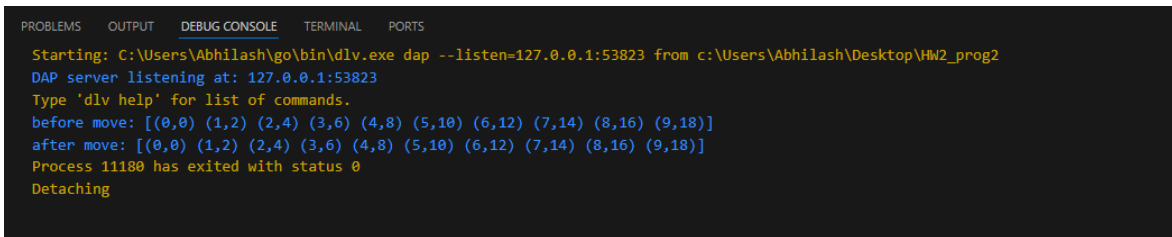
Code 5: The given main source code for the second program.

### 2.1 Running the code

Running the code in a shared workspace directory of that of program 1 negates any run-time error pertaining to initialization or running of the Go module. However, for the purposes of testing, the program 2 was run in a different directory and the same error that had popped up for program 1 shown in Figure 1 was seen again. There after following the same steps as done for program 1 demonstrated above in Code 4. Although the *json* launch file was not required once more even after re-launching VS Code to refresh the instance after program 1.

#### 2.1.1 Existing Condition

The code run in its existing condition provides the following output as shown in Figure 3.

The image shows a terminal window with a dark background and light-colored text. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL', and 'PORTS'. The 'TERMINAL' tab is active. The output text is as follows:

```
Starting: C:\Users\Abhilash\go\bin\dlv.exe dap --listen=127.0.0.1:53823 from c:\Users\Abhilash\Desktop\HW2_prog2
DAP server listening at: 127.0.0.1:53823
Type 'dlv help' for list of commands.
before move: [(0,0) (1,2) (2,4) (3,6) (4,8) (5,10) (6,12) (7,14) (8,16) (9,18)]
after move: [(0,0) (1,2) (2,4) (3,6) (4,8) (5,10) (6,12) (7,14) (8,16) (9,18)]
Process 11180 has exited with status 0
Detaching
```

Figure 3: Initial unaltered program output.

It is to note that from Figure 3, the program runs but does not satisfy the operation in performing as it should to solve the problem. It can be observed that the move operation from the vertices before and after, have not moved or changed at all.

### 2.1.2 Updated Condition

The error was addressed by making the following correction to the program and the corrected code is mentioned in Code 6.

```
package main

import (
    "fmt"
)

type Vertex struct {
    X, Y int
}

func (v *Vertex) Move(dx, dy int) {
    v.X += dx
    v.Y += dy
}

func (v Vertex) String() string {
    return fmt.Sprintf("(%d,%d)", v.X, v.Y)
}

func main() {
    a := make([]Vertex, 0)
    for i := 0; i < 10; i++ {
        a = append(a, Vertex{X: i, Y: i * 2})
    }
    fmt.Printf("before move: %v\n", a)
    for i := range a {
        a[i].Move(1, 2)
    }
    fmt.Printf("after move: %v\n", a)
}
```

Code 6: Code snippet for the updated code from the original code.

After having updated the code, the following Figure 4 illustrates the resultant output thereby indicating that the program is behaving as it should and providing correct results. This can be observed by looking at the before and after of the move operation and there is a shift between the same.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Starting: C:\Users\Abhilash\go\bin\dlv.exe dap --listen=127.0.0.1:53847 from c:\Users\Abhilash\Desktop\HW2_prog2
DAP server listening at: 127.0.0.1:53847
Type 'dlv help' for list of commands.
before move: [(0,0) (1,2) (2,4) (3,6) (4,8) (5,10) (6,12) (7,14) (8,16) (9,18)]
after move: [(1,2) (2,4) (3,6) (4,8) (5,10) (6,12) (7,14) (8,16) (9,18) (10,20)]
Process 12596 has exited with status 0
Detaching
```

Figure 4: Correct and updated results from update to the code.

## 2.2 Explanation

The original snippet of code is as below in Code 7.

```
for _, v := range a {
    v.Move(1, 2)
}
```

Code 7: Original non edited portion of code.

The initial slice elements are not altered by this loop, contrary to what one may assume. This is the reason why:

1. Range Loop Operation: The loop variables (in this example, '\_' for the index and 'v' for the value) are copies of the slice elements when you use a 'for... range' loop on a slice; they are not pointers to the original items.
2. Value semantics: Every iteration starts with a new variable called 'v', which holds a duplicate copy of the slice's 'vertex'. Changes made to 'v' only impact this copy and not the slice's original 'vertex'.
3. Receiver Method: Calling 'v.Move(1, 2)' still uses the copy of the 'vertex', not the one in the slice, even though the 'Move' function has a pointer recipient '(v \*vertex)'.

Therefore, copies of the vertices receive a call to the Move function, but the original slice stays unaltered.

The fixed code snippet is as below in Code 8.

```
for i := range a {
    a[i].Move(1, 2)
}
```

Code 8: Updated edited portion of code.

Go's architecture has many important features that allow the new technique to modify the slice parts effectively:

1. Straight Slice Modification and Access: We may directly access and alter the 'vertex' kept in the slice by using 'a[i]'. This is made feasible by the fact that in Go, a slice is just a struct that has length, capacity, and a reference to an underlying array. When we change 'a[i]', we're changing an element in this underlying array, and the slice reflects this change.

2. Automatic Pointer Dereferencing: Go automatically uses the address of `'a[i]'` to match the pointer recipient of the `'Move'` function `'(func (v *Vertex) Move(dx, dy int))'` when we call `'a[i].Move(1, 2)'`. We can be confident we're changing the real `Vertex` in the slice and not a replica thanks to this automated dereferencing.
3. Slice Internals and Method Receiver Matching: The internal workings of slices, along with Go's method receiver matching mechanism, make it possible to modify slice elements in an effective and user-friendly manner. Go is intelligent enough to understand that the address of a value `'(a[i])'` should be passed to a pointer receiver method (such as `'Move'`) when it is called.

Value-based range loops are frequently not as effective as index-based iteration when it comes to changing slice components inside of a loop. By doing this, you can be confident that you're dealing with the slice's original elements rather than duplicates.

This idea is essential to comprehending how method receivers work with various kinds of loop variables and how Go manages data in loops. It's a frequent cause of subtle problems in Go applications, particularly for inexperienced programmers.