Solution Review: Reverse a String

This lesson discusses the solution to the challenge given in the previous lesson.

WE'LL COVER THE FOLLOWING

- 1st Variant
- 2nd Variant
- 3rd Variant

```go
package main
import "fmt"

// variant: use of slice of byte and conversions
func reverse1(s string) string {
	sl := []byte(s)
	var rev [100]byte
	j := 0
	for i:=len(sl)-1; i >= 0; i-- {
		rev[j] = sl[i]
		j++
	}
	strRev := string(rev[:len(sl)])
	return strRev
}

// variant: "in place" using swapping _one slice
func reverse2(s string) string {
	sl2 := []byte(s)
	for i, j := 0, len(sl2) - 1; i < j; i, j = i+1, j-1 {
		sl2[i], sl2[j] = sl2[j], sl2[i]
	}
	return string(sl2)
}

//variant: using [] int for runes (necessary for Unicode-strings!):
func reverse3(s string) string {
	runes := []rune(s)
	n, h := len(runes), len(runes)/2
	for i:= 0; i < h; i ++ {
		runes[i], runes[n-1-i] = runes[n-1-i], runes[i]
	}
	return string(runes)
}

func main() {
// reverse a string:
```

```
// reverse a string:
        str := "Google"

        fmt.Printf("The reversed string using variant 1 is -%s-\n", reverse1(str))
        fmt.Printf("The reversed string using variant 2 is -%s-\n", reverse2(str))
        fmt.Printf("The reversed string using variant 3 is -%s-\n", reverse3(str))
}
```

Reverse a String

In the above code, there are *three* basic functions: `reverse1`, `reverse2`, and `reverse3`. Let's study them one by one.

1st Variant

Look at the header of function `reverse1` at **line 5**: `func reverse1(s string) string`. It takes a string `s` as a parameter to reverse it and returns the reversed string. This variant is the implementation of a slice of bytes, converting it later into a string.

At **line 6**, we make a slice of bytes `sl`, containing all the elements from string `s` passed to the function `reverse1`. Next, we make an array `rev` of byte type of length **100** (assuming that the length of the string `s` doesn't exceed **100**). At **line 8**, we make an *iterator* `j` that we will use to traverse through the `rev` array as we move forward, filling `rev` with values.

Then we have a for loop at **line 9**. You may have noticed that the iterator `i` of this loop is initialized with `len(sl)-1`. It will move in a backward direction until we reach the **0th** index. This is because we have to reverse the string; the idea is to read `sl` in a backward direction and store each character in `rev` in the forward direction. That's the reason we have a separate iterator `j` for `rev` that starts from **0**. At **line 10**, we set `rev[j]` as `sl[i]` for each iteration and increment `j` by **1** at **line 11** to move one index forward in `rev`. By the end of the for loop, we'll have the reversed string in `rev`.

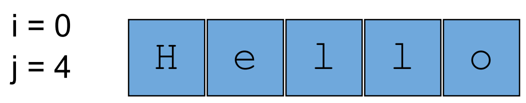
But, there is one problem. Remember we set the length of `rev` to 100 bytes. What if all the bytes are not filled with values and are *nill*? For this, you have to reslice `rev` as: `rev[:len(sl)]` (see **line 13**) because it is obvious that the actual length of `rev` is equal to `sl`. Don't forget to convert the `rev` type to *string*, before returning it. The `reverse1` returns a *string* type variable.

2nd Variant

Now, look at the header of the second variant `reverse2` at **line 18**: `func reverse2(s string) string`. It takes a string `s` as a parameter to reverse it and returns the reversed string. This variant is the implementation of using just one slice of bytes, and it does in-place swapping, unlike `reverse1`, converting it later into a string.

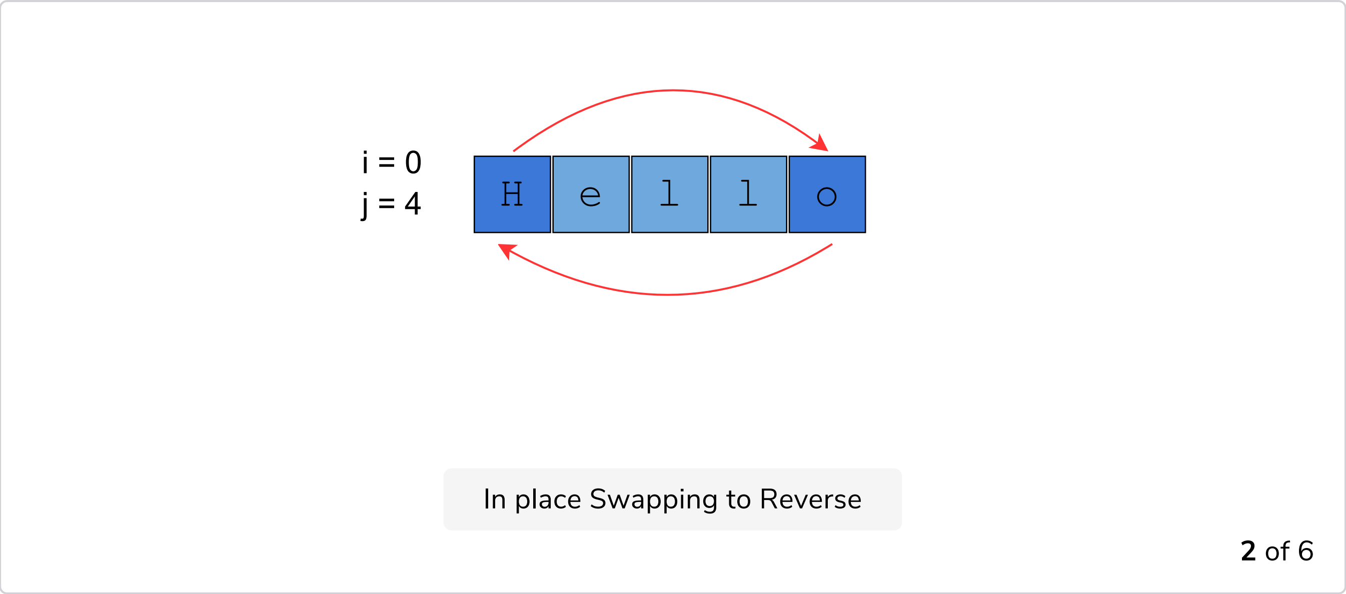
At **line 19**, we make a slice of bytes `sl2`, containing all the elements from string `s` passed to the function `reverse2`. Then, we have a for loop at **line 20**. As we are doing in-place swapping, we make two iterators `i` and `j` in a single for loop. The idea is the same. One iterator will move in the forward direction where the second iterator will move in the backward direction. You may have noticed that iterator `i` of this loop is initialized with **0** which means it will move in the forward direction and will change the values of `sl2`. The iterator `j` is initialized with `len(sl2)-1` and will move in the backward direction because it will read the values from `sl2`, which are to be placed by `i` in the opposite direction.

We'll do this in-place swapping until `i` and `j` don't coincide which means: `i<j`. Because when `i` is equal to `j`, it means the string is reversed, as `i` and `j` are incremented and decremented by **1** respectively. However, the question is how to do in-place swapping? Look at **line 21**, we set `sl2[i]` as `sl2[j]`, and `sl2[j]` as `sl2[i]`. Let's visualize it. Suppose we have a string of length **5**. Let's do some in-place swapping:



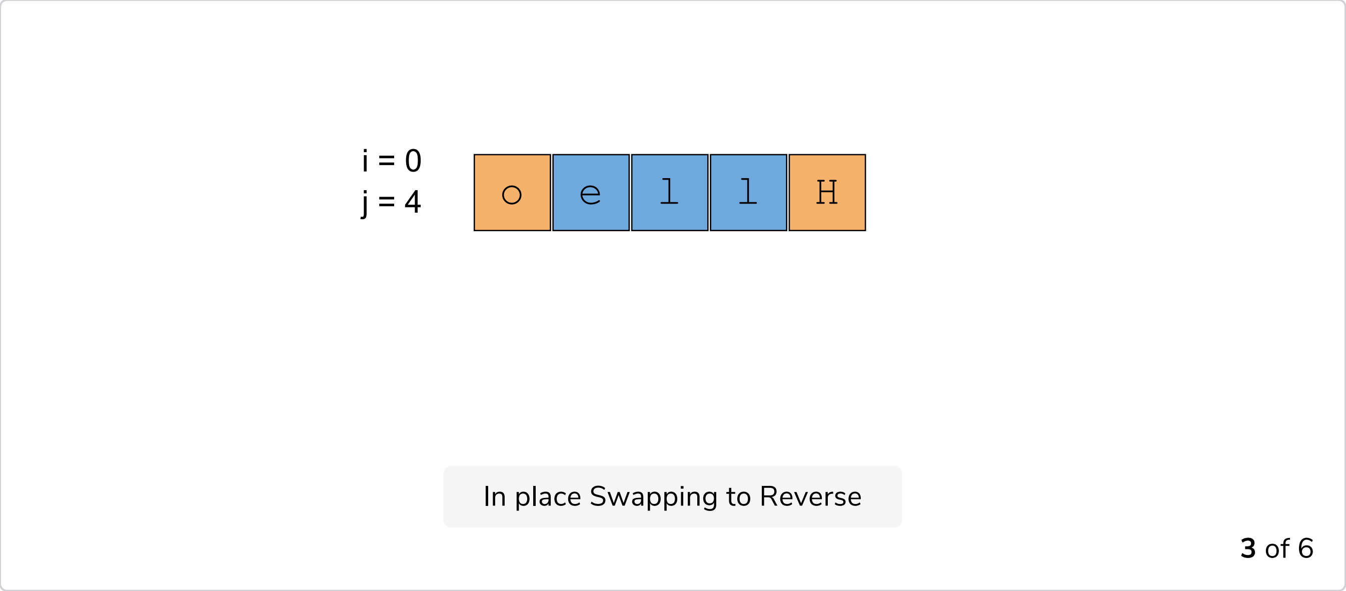
In place Swapping to Reverse

1 of 6



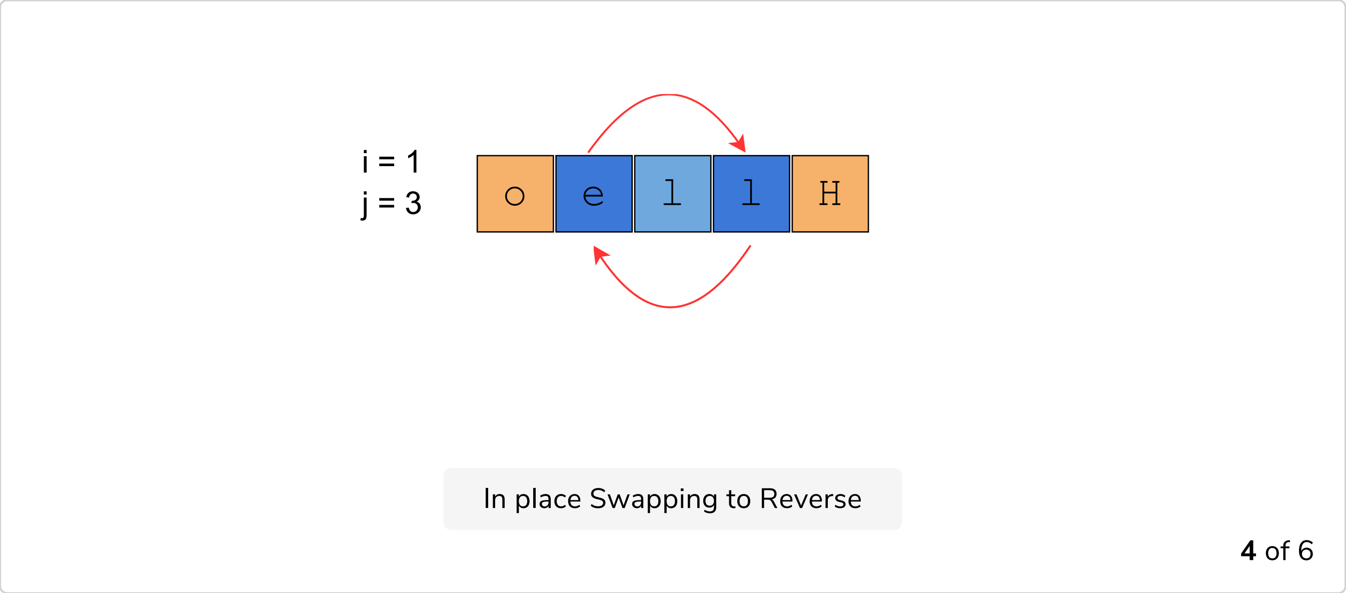
In place Swapping to Reverse

2 of 6



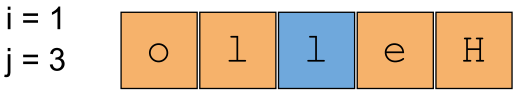
In place Swapping to Reverse

3 of 6



In place Swapping to Reverse

4 of 6

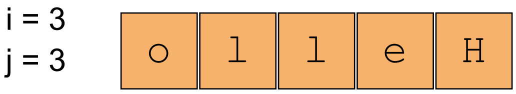


i = 1
j = 3

o	l	l	e	H

In place Swapping to Reverse

5 of 6



i = 3
j = 3

o	l	l	e	H

In place Swapping to Reverse

6 of 6

By the end of the for loop, we'll have the reversed string in `s12`. Don't forget to convert the `s12`'s type from *byte* to *string* before returning it because `reverse2` returns a *string* type variable.

3rd Variant

Now, look at the header of the third variant `reverse3` at **line 27**: `func reverse3(s string) string`. It takes a string `s` as a parameter to reverse and returns the reversed string. This variant is the implementation of using `[]int` for `runes`, in case the string is a Unicode string, converting rune later into a string.

At **line 28**, we make a slice of *runes* called `runes`, containing all the elements from string `s` passed to the function `reverse3`. In the next line (**line 29**), we from string `s` passed to the function `reverse3`. In the next line (**line 29**), we declare two variables `n` and `h`. We set `n` as the length of runes: `len(runes)` and `h` as half the length of `runes`: `len(runes)/2`. Now, we'll conduct in-place swapping just like we did in the case of `reverse2` but with a single iterator. We make a for loop at **line 30**. We make an iterator `i` that starts with **0** and will move until the `h-1` index (half the length of `runes`).

You may have noticed that in in-place swapping we only have to forward half the length of the string. So the iterator `i` will be incremented by **1** after each iteration. However, the question is how to do in-place swapping with just one iterator? Remember we made a variable `n` at **line 29** previously. We'll use a combination of `i` and `n` in this variant. Look at **line 31**. It's obvious that after reversing a string the first character comes in the last place and the last character comes in the first place. We set `runes[i]` to `runes[n-1-i]`, and `runes[n-1-i]` to `runes[i]`.

Let's suppose we have a string of length **5**, so `n` would be **5**. In the first iteration, `i` would be **0**. So, `runes[0]` would be equal to `runes[4]`, which means that the first element now has the value of the last element, and `runes[4]` would be equal to `runes[0]`, which means that the last element now has the value of the first element. **Line 31** will run for each iteration until `i` equals `h`. By the end of the for loop, we'll have the reversed string in `runes`. Don't forget to convert *runes*' type to *string* before returning it because `reverse3` returns a *string* type variable.

Let's see `main` now. In `main`, at **line 38**, we made a string `str` and set its value to **Google**. We'll reverse this string, to see whether our three variants work or not. We call each of the variants in the `main` to verify the results. **Line 40** prints the string returned by `reverse1`. **Line 41** prints the string returned by `reverse2`. **Line 42** prints the string returned by `reverse3`. These three statements print **elgooG**, which is the reversed form of **Google**.

That's it about the solution. There is a quiz in the next lesson for you to solve.