Chapter 18 슬라이스

날짜	@December 1, 2021
발표자	김하연

18.1 슬라이스

18.1.1 슬라이스 선언

일반 배열은 처음 배열을 만들 때 정한 길이에서 더 이상 늘어나지 않는다.

```
var array [10]int
```

- 최대 10개까지 int 값 저장
- 더 많은 값 저장? 더 큰 배열 → 값을 하나씩 복사

슬라이스: 배열과 비슷하지만 [] 안에 배열의 개수를 적지 않고 선언

```
var slice []int
```

슬라이스 초기화하지 않으면 길이가 0인 슬라이스, 길이 초과 접근하면 런타임 에러 발생

```
package main

import "fmt"

func main() {
  var slice []int

  if len(slice) == 0 {
    fmt.Println("slice is empty", slice)
  }
```

```
  slice[1] = 10
  fmt.Println(slice)
}
```

출력문

```
slice is empty []
panic: runtime error: index out of range [1] with length 0

goroutine 1 [running]:
main.main()
        C:/Users/WINDOWS10/Desktop/Golang-Study/Ch18/ex18.1/ex18.1.go:12 +0x87
```

- slice 길이 검사 → 길이 0 : slice[1] 접근 시 패닉 발생(비정상 종료)

{}를 이용해 초기화하기

배열처럼 {}를 사용해서 요솟값 지정

```
var slice1 = []int{1, 2, 3}
var slice2 = []int{1, 5:2, 10:3}  // {1 0 0 0 0 2 0 0 0 0 3}

var array = [...]int{1, 2, 3} //길이 3인 배열
var slice = []int{1, 2, 3}    //슬라이스
```

make()를 이용한 초기화

make() 함수의 첫 번째 인수: 만들고자 하는 타입, 두 번째 인수: 길이

```
var slice = make([]int, 3)
```

길이 3인 int 슬라이스 값, 각 요솟값은 기본값

18.1.2 슬라이스 요소 접근

접근 방법은 배열과 똑같다.

```
var slice = make([]int, 3)
slice[1] = 5
```

18.1.3 슬라이스 순회

순회 역시 배열과 똑같다. (동적으로 길이가 늘어나는 점만 제외하면)

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

for i := 0; i < len(slice); i++ {
  slice[i] += 10
}

for i, v := range slice {
  slice[i] = v * 2
}
```

- len() 함수로 slice 길이 알아내기 → 순회
- range 키워드 → 순회 (range의 값: 인덱스, 요솟값)

18.1.4 슬라이스 요소 추가 - append()

슬라이스만의 기능인 요소를 추가하는 방법!

기존 배열은 한 번 길이가 정해지면 늘릴 수 없지만 슬라이스는 요소를 추가해 길이를 늘릴 수 있다.

append() 함수 첫 번째 인수: 추가하고자 하는 슬라이스, 두 번째 인수 요소 → 슬라이스 맨 뒤에 요소를 추가해 새로운 슬라이스를 결과로 반환

```
package main

import "fmt"

func main() {

  var slice = []int{1, 2, 3}

  slice2 := append(slice, 4)

  fmt.Println(slice)
  fmt.Println(slice2)
}
```

출력문

```
[1 2 3]
[1 2 3 4][1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
```

- slice의 길이: 3
- append() 함수 사용 → 맨 뒤에 요솟값 4를 추가해 반환한 슬라이스 slice2에 대입
- slice2의 길이: 4

18.1.5 여러 값 추가

append()를 사용해 값을 하나 이상 추가할 수 있다.

```
slice = append(slice, 3, 4, 5, 6, 7)
```

두 번째 인수 이후로 추가하고 싶은 값들을 적어준다. append()는 첫 번째 인수로 들어온 슬라이스의 값을 변경하는 게 아니라 요소가 추가된 새로운 슬라이스를 반환한다 → 기존 슬라이스에 추가? 기존 슬라이스에 대입

```
package main

import "fmt"

func main() {
  var slice []int

  for i := 1; i <= 10; i++ {
    slice = append(slice, i)
  }

  slice = append(slice, 11, 12, 13, 14, 15)
  fmt.Println(slice)
}
```

출력문

```
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
```

- for문 사용: 1~10 하나씩 추가

- append() 여러 값: 11~15 한 번에 추가

슬라이스는 이와 같이 배열과 사용법이 비슷하지만 동적 배열로 동작한다. 하지만 중요한 차이점이 있다.

18.2 슬라이스 동작 원리

슬라이스의 원리를 제대로 이해하지 않으면 예기치 못한 버그가 발생할 수 있다.

슬라이스는 내장 타입 → reflect 패키지의 SliceHeader 구조체로 내부 구현 살펴보기

```
type SliceHeader struct{
  Data  uintptr   //실제 배열을 가리키는 포인터
  Len   int       //요소 개수
  Cap   int       //실제 배열의 길이
}
```

- 슬라이스가 실제 배열을 가리키는 포인터를 가지고 있어서 쉽게 크기가 다른 배열을 가리키도록 변경 가능할 수 있고, 슬라이스 변수 대입 시 배열에 비해서 사용되는 메모리나 속도에 이점이 있다!

18.2.1 make() 함수를 이용한 선언

```
var slice = make([]int, 3)
```

slice는 len이 3, cap이 3 = 총 배열 길이가 3, 요소 개수가 3.

```
var slice2 = make([]int, 3, 5)
```

slice2는 len이 3, cap이 5 = 총 배열 길이가 5, 요소 개수가 3. 총 5개 중 3개만 사용하고 나머지 2개는 나중에 추가될 요소를 위해 비워두었다.

18.2.2 슬라이스와 배열의 동작 차이

배열과 내부 구현이 다르기 때문에 동작도 매우 다르다! 똑같이 사용하면 예기치 못한 버그를 만날 수 있다!

```
package main

import "fmt"

func changeArray(array2 [5]int) {
  array2[2] = 200
}

func changeSlice(slice2 []int) {
  slice2[2] = 200
}

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

  changeArray(array)
  changeSlice(slice)

  fmt.Println("array:", array)
  fmt.Println("slice:", slice)
}
```

출력문

```
array: [1 2 3 4 5]
slice: [1 2 200 4 5]
```

- changeArray() 함수: 배열을 매개변수로, 3번째 값 → 200
- changeSlice() 함수: 슬라이스를 매개변수로, 3번째 값 → 200
- slice의 값은 200으로 바뀌었지만, array의 값은 바뀌지 않았다! 왜???

18.2.3 동작 차이의 원인

Go 언어에서 모든 값의 대입은 복사로 일어난다. 함수의 인수 전달, 변수의 대입, 값의 이동 등 복사. 복사는 타입의 값이 복사된다. 포인터의 메모리 주소, 구조체의 모든 필드, 배열의 모든 값이 복사된다.

chageArray() 함수

array ([5]int 타입, 40바이트) → 모든 값 복사 → array2

array와 array2는 메모리 공간이 다른, 즉 완전히 다른 배열이다. array2의 3번째 값 변경 → array 영향 X

chageSlice() 함수

[]int 타입: 포인터(8바이트), len(8바이트), cap(8바이트) ⇒ 총 24바이트

slice([]int 타입, 24바이트) → 모든 필드 복사 → slice2

slice와 slice2는 똑같은 메모리 주솟값을 가지기 때문에 같은 배열 데이터를 가리키게 된다. 따라서 slice2의 3번째 값 변경 → slice의 3번째 값 변경

18.2.4 append()를 사용할 때 발생하는 예기치 못한 문제 1

append() 함수 호출: 슬라이스에 값을 추가할 수 있는 빈 공간이 있나?

```
남은 빈 공간 = cap - len
```

남은 빈 공간의 개수가 추가하는 값의 개수보다 크거나 같은 경우 배열 뒷부분에 값을 추가한 뒤 len값 증가

slice1 = Data: {1, 2, 3, , }, len: 3, cap: 5

```
slice2 := append(slice1, 4, 5)
```

slice1에 빈 공간 있나? cap - len = 2 → 4, 5 추가 가능, len값 2 증가시킨 슬라이스 반환

slice2 = Data{1, 2, 3, 4, 5}, len: 5, cap: 5

slice1와 slice2는 같은 배열을 가리킨다. 요소 개수는 각 3, 5.

```
slice1[1] = 100
```

slice1와 slice2 배열의 두 번째 값 100으로 변경

```
slice1 = append(slice1, 500)
```

slice1에 빈 공간 있나? cap - len = 2 → slice1[len] 자리에 500, len 1 증가시킨 슬라이스 반환

배열 {1, 100, 3, 500, 5}

slice1 = len: 4, cap: 5

slice2 = len: 5, cap: 5

```
package main

import "fmt"

func main() {
  slice1 := make([]int, 3, 5) // len:3 cap:5

  slice2 := append(slice1, 4, 5)
  fmt.Println("slice1:", slice1, len(slice1), cap(slice1))
  fmt.Println("slice2:", slice2, len(slice2), cap(slice2))

  slice1[1] = 100

  fmt.Println("After change second element")
  fmt.Println("slice1:", slice1, len(slice1), cap(slice1))
  fmt.Println("slice2:", slice2, len(slice2), cap(slice2))

  slice1 = append(slice1, 500)

  fmt.Println("After append 500")
  fmt.Println("slice1:", slice1, len(slice1), cap(slice1))
  fmt.Println("slice2:", slice2, len(slice2), cap(slice2))
}
```

출력문

```
slice1: [0 0 0] 3 5
slice2: [0 0 0 4 5] 5 5
After change second element
slice1: [0 100 0] 3 5
slice2: [0 100 0 4 5] 5 5
After append 500
slice1: [0 100 0 500] 4 5
slice2: [0 100 0 500 5] 5 5
```

18.2.5 append()를 사용할 때 발생하는 예기치 못한 문제 2

만약 빈 공간이 없다면?

append() 함수 호출 → 빈 공간이 충분한가?

만약 빈 공간이 충분하지 않으면

- 새로운 더 큰 배열을 마련한다. (일반적으로 기존 배열의 2배 크기) 기존 배열의 요소를 모두 새로운 배열에 복사, 맨 뒤에 새 값 추가.
- 반환하는 슬라이스 = cap: 새로운 배열의 길이, len: 기존 길이에 추가한 개수만큼 더한 값, 포인터: 새로운 배열을 가리킴

```
slice1 := []int{1, 2, 3}
slice2 := append(slice1, 4, 5)
```

slice1 = len: 3, cap: 3 (빈 공간 X)

길이 6인 새로운 배열 → slice 값 복사 & 4, 5 추가 → len: 4, cap: 6 슬라이스 반환 → slice2 대입

slice1과 slice2는 다른 배열을 가리킴.

```
slice1[1] = 100
```

slice1의 두 번째 값만 100으로 변함. (slice2는 다른 배열을 가리킴)

```
slice1 = append(slice1, 500)
```

역시 slice2에 영향 X

```
package main

import "fmt"

func main() {
  slice1 := []int{1, 2, 3} // len:3 cap:3

  slice2 := append(slice1, 4, 5)

  fmt.Println("slice1:", slice1, len(slice1), cap(slice1))
  fmt.Println("slice2:", slice2, len(slice2), cap(slice2))

  slice1[1] = 100

  fmt.Println("After change second element")
  fmt.Println("slice:", slice1, len(slice1), cap(slice1))
```

```
  fmt.Println("slice2:", slice2, len(slice2), cap(slice2))

  slice1 = append(slice1, 500)

  fmt.Println("After append 500")
  fmt.Println("slice1:", slice1, len(slice1), cap(slice1))
  fmt.Println("slice2:", slice2, len(slice2), cap(slice2))
}
```

출력문

```
slice1: [1 2 3] 3 3
slice2: [1 2 3 4 5] 5 6
After change second element
slice: [1 100 3] 3 3
slice2: [1 2 3 4 5] 5 6
After append 500
slice1: [1 100 3 500] 4 6
slice2: [1 2 3 4 5] 5 6array: [1 2 3 4 5]
slice: [2] 1 4
After change second element
array: [1 100 3 4 5]
slice: [100] 1 4
After append 500
array: [1 100 500 4 5]
slice: [100 500] 2 4
```

18.3 슬라이싱

슬라이싱: 배열의 일부를 집어내는 기능, 슬라이스 반환

```
array[startIdx:endIndex]
```

대상이 되는 배열, 대괄호 [ ] 사이에 집어내고자 하는 **시작인덱스 : 끝인덱스** → **시작인덱스**부터 **끝인덱스-1**까지의 배열 일부를 나타내는 슬라이스 반환 (주의: 끝인덱스 포함 X, 새로운 배열이 만들어지는 게 아니라 일부를 포인터로 가리키는 슬라이스)

```
package main

import "fmt"

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

```
  slice := array[1:2]

  fmt.Println("array:", array)
  fmt.Println("slice:", slice, len(slice), cap(slice))

  array[1] = 100

  fmt.Println("After change second element")
  fmt.Println("array:", array)
  fmt.Println("slice:", slice, len(slice), cap(slice))

  slice = append(slice, 500)

  fmt.Println("After append 500")
  fmt.Println("array:", array)
  fmt.Println("slice:", slice, len(slice), cap(slice))
}
```

출력문

```
array: [1 2 3 4 5]
slice: [2] 1 4
After change second element
array: [1 100 3 4 5]
slice: [100] 1 4
After append 500
array: [1 100 500 4 5]
slice: [100 500] 2 4
```

- array[1:2] → [1]만 집어낸 슬라이스 slice에 대입
- array[1] 변경 → slice값 변경
- 슬라이싱할 때 cap길이는 array의 인덱스 1에서부터 배열의 마지막 인덱스까지의 길이

18.3.1 슬라이싱으로 배열 일부를 가리키는 슬라이스 만들기

슬라이스는 배열의 일부를 나타내는 타입. 포인터: 얼마든지 배열의 중간을 가리킬 수 있음. len: 포인터가 가리키는 메모리부터 일정 개수. cap: 포인터가 가리키는 배열이 할당된 크기 즉 안전하게 사용 가능한 남은 배열 개수

```
array := [5]int{1, 2, 3, 4, 5}
slice := array[1:2]
```

slice의 포인터: 배열의 시작인덱스 1 → 두 번째 요소 메모리 주소

slice의 len: 끝인덱스 - 시작인덱스 = 2 - 1 = 1

slice의 cap: 배열 총길이 - 시작인덱스 = 5 - 1 = 4

빈 공간: 4 - 1 = 3

array의 두 번째 값이 변하면 slice값도 바뀐다.

```
array[1] = 100
```

slice에 요소 추가 → 빈 공간 있음 → array[2] 변경

```
slice = append(slice, 500)
```

18.3.2 슬라이스를 슬라이싱하기

배열뿐 아니라 슬라이스 일부를 집어낼 수 있다.

```
slice1 := []int{1, 2, 3, 4, 5}
slice2 := slice[1:2]
```

slice2: slice1이 가리키는 배열의 시작인덱스 1인 두 번째 요소를 가리키는 슬라이스, len: 1, cap: 4

처음부터 슬라이싱

```
slice1 := []int{1, 2, 3, 4, 5}
slice2 := slice1[0:3]
```

- slice2: slice1의 첫 번째~세번째까지, [1, 2, 3] 세 요소를 가짐. cap: 5 - 0 = 5
- 첫번째부터 슬라이싱 → 시작인덱스 생략 가능

```
slice2 := slice1[0:3]
slice2 := slice1[:3]
```

끝까지 슬라이싱

```
slice1 := []int{1, 2, 3, 4, 5}
slice2 := slice1[2:len(slice1)]
```

- slice2: slice1의 세번째~끝까지, [3, 4, 5] 세 요소를 가짐. cap: 5 - 2 = 3
- 끝까지 슬라이싱 → 끝인덱스 생략 가능

```
slice2 := slice[2:len(slice1)]
slice2 := slice[2:]
```

전체 슬라이싱

```
array := [5]int{1, 2, 3, 4, 5}
slice := array[:]
```

처음부터 끝까지 슬라이싱 → 시작인덱스, 끝인덱스 생략

인덱스 3개로 슬라이싱해 cap 크기 조절하기

인덱스 2개만 사용, cap은 배열의 전체 길이 - 시작인덱스. 인덱스 3개 사용해서 cap까지 조절 가능!

```
slice[시작인덱스 : 끝인덱스 : 최대인덱스]
```

시작인덱스부터 끝인덱스 하나 전까지 집어내고 최대인덱스까지만 배열 사용 → cap: 최대인덱스 - 시작인덱스

```
slice1 := []int{1, 2, 3, 4, 5}
slice2 := slice1[1:3:4]
```

인덱스 1부터 2까지 집어내기 → [2, 3], cap: 4 - 1 = 3

슬라이싱할 때 세번째 인덱스 X : 배열의 전체 길이, 세번째 인덱스 O : 그 인덱스까지만 배열 사용

18.4 유용한 슬라이싱 기능 활용

슬라이싱과 append() 기능 → 슬라이싱 복제, 요소 추가, 요소 삭제

18.4.1 슬라이스 복제

두 슬라이스가 서로 같은 배열을 가리켜서 발생하는 문제 → 항상 다른 배열을 가리켜서 문제 없도록? 슬라이스 복제하기

```
package main

import "fmt"

func main() {
  slice1 := []int{1, 2, 3, 4, 5}

  slice2 := make([]int, len(slice1))

  for i, v := range slice1 {
    slice2[i] = v
  }

  slice1[1] = 100
  fmt.Println(slice1)
  fmt.Println(slice2)
}
```

출력문

```
[1 100 3 4 5]
[1 2 3 4 5]
```

- slice1과 똑같은 길이의 다른 슬라이스 생성 & 모든 요솟값 복사
- slice1의 요솟값 변경 → slice2 영향 X

append() 함수로 코드 개선하기

같은 길이의 슬라이스 생성 & 순회로 각 요솟값 복사 → 한 줄로

```
slice2 := append([]int{}, slice1...)
```

append() 함수 사용 → slice1의 모든 값 복제한 새로운 슬라이스 slice2에 대입 (배열이나 슬라이스 뒤에 ...를 하면 모든 요솟값을 넣어준 것과 같다)

```
slice2 := append([]int{}, slice1[0], slice1[1], slice1[2], slice1[3], slice1[4])
```

copy() 함수로 코드 개선하기

```
func copy(dst, src []Type) int
```

copy() 함수: 첫 번째 인수로 복사한 결과를 저장하는 슬라이스 변수, 두 번째 인수로 복사 대상이 되는 슬라이스 변수, 반환값은 실제로 복사된 요소 개수

실제 복사되는 요소 개수는 목적지의 슬라이스 길이와 대상의 슬라이스 길이 중 작은 개수만큼 복사된다.

```
package main

import "fmt"

func main() {
  slice1 := []int{1, 2, 3, 4, 5}
  slice2 := make([]int, 3, 10)
  slice3 := make([]int, 10)

  cnt1 := copy(slice2, slice1)
  cnt2 := copy(slice3, slice1)

  fmt.Println(cnt1, slice2)
  fmt.Println(cnt2, slice3)
}
```

출력문

```
3 [1 2 3]
5 [1 2 3 4 5 0 0 0 0 0][1 2 4 5 6]
```

- slice2 = len: 3, cap: 10
- slice3 = len: 10, cap: 10

- slice1을 slice2에 복사 → slice1의 요소 5개, slice2의 요소 3개 → 3개 복사 (cap 개수 영향 X)
- slice1을 slice3에 복사 → slice1의 요소 5개, slice3의 요소 10개 → 5개 복사

```
slice2 := make([]int, len(slice1))
copy((slice2, slice1)
```

먼저 같은 길이의 슬라이스 생성 후 복사

18.4.2 요소 삭제

슬라이스 중간의 요소 삭제하는 방법: 중간 요소 삭제, 중간 요소 이후의 값을 앞당겨서 삭제되 요소 채우기, 맨 마지막값 지우기

```
package main

import "fmt"

func main() {
  slice := []int{1, 2, 3, 4, 5, 6}
  idx := 2 // 삭제할 인덱스

  for i := idx + 1; i < len(slice); i++ {
    slice[i-1] = slice[i]
  }

  slice = slice[:len(slice)-1]

  fmt.Println(slice)
}
```

출력문

```
[1 2 4 5 6]
```

- 삭제 위치 3번째 → 4번째부터 앞당기기 → [1, 2, 4, 5, 6, 6] → 마지막 요소 잘라내기

append() 함수로 코드 개선하기

```
slice = append(slice[:idx], slice[idx+1]...)
```

slice[:idx]는 처음부터 idx 전까지 집어낸 슬라이스.

slice[idx+1:]는 idx 하나 뒤의 값부터 끝까지 집어낸 슬라이스

append() 함수로 두 슬라이스 붙이기

18.4.3 요소 추가

슬라이스 중간에 요소 추가하는 방법 : 슬라이스 맨 뒤에 요소 하나 추가 → 맨 뒤값부터 삽입하려는 위치까지 한 칸씩 뒤로 밀어주기 → 삽입하는 위치의 값 바꿔주기

```
package main

import "fmt"

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

  slice = append(slice, 0)  // 맨 뒤에 요소 추가

  idx := 2 // 추가하려는 위치

  for i := len(slice) - 2; i >= idx; i-- {
    slice[i+1] = slice[i]
  }

  slice[idx] = 100

  fmt.Println(slice)
}
```

출력문

```
[1 2 100 3 4 5 6]
```

- 맨 뒤에 요소 추가 → 맨 뒤부터 삽입하는 자리까지 한씩 뒤로 밀기 → idx 위치의 값 바꾸기

append() 함수로 코드 개선하기

```
slice = append(slice[:idx], append([]int{100}, slice[idx:]...)...)
```

append() 중첩으로 사용

slice[:idx]는 처음부터 idx 전까지 집어낸 슬라이스.

[]int{100}은 삽입하려는 값 100 한 개만 갖는 슬라이스.

slice[idx:]는 idx부터 끝까지 집어낸 슬라이스

append() 함수로 삽입하려는 값을 가지는 슬라이스 + idx부터 끝까지의 슬라이스

append() 함수로 처음부터 idx 전까지의 슬라이스 + (삽입하려는 슬라이스 + idx부터 끝까지의 슬라이스)

불필요한 메모리 사용이 없도록 코드 개선하기

앞의 구문은 임시 슬라이스 사용 → 불필요한 메모리

```
slice = append(slice, 0)          // 맨 뒤에 요소 추가
copy(slice[idx+1:], slice[idx:])  // 값 복사
slice[idx] = 100                  // 값 변경
```

- slice 맨 뒤에 요소 추가 → copy() 사용해 슬라이스값 복사 (idx 하나 다음부터 끝까지를 idx부터 복사) → idx 위치에 100 대입

18.5 슬라이스 정렬

Go언어에서 기본 제공하는 sort 패키지 사용해 슬라이스 정렬하기

18.5.1 int 슬라이스 정렬

```
package main

import (
  "fmt"
  "sort"
)

func main() {
  s := []int{5, 2, 6, 3, 1, 4}
  sort.Ints(s)
  fmt.Println(s)
}
```

출력문

```
[1 2 3 4 5 6]
```

- sort 패키지의 Ints() 함수 → []int 슬라이스 정렬
- Float64s() 함수 → float64 슬라이스 정렬

18.5.2 구조체 슬라이스 정렬

Sort() 함수 → Len(), Less(), Swap() 메서드 필요

```
package main

import (
  "fmt"
  "sort"
)

type Student struct {
  Name string
  Age  int
}

type Students []Student

func (s Students) Len() int           { return len(s) }
func (s Students) Less(i, j int) bool { return s[i].Age < s[j].Age }
func (s Students) Swap(i, j int)      { s[i], s[j] = s[j], s[i] }

func main() {
  s := []Student{
    {"화랑", 31}, {"백두산", 52}, {"류", 42},
    {"켄", 38}, {"송하나", 18}}

  sort.Sort(Students(s))
  fmt.Println(s)
}
```

출력문

```
[{송하나 18} {화랑 31} {켄 38} {류 42} {백두산 52}]
```

- 구조체 슬라이스를 나이순으로 정렬
- []Student의 별칭 타입 Students 생성

- Len(), Less(), Swap() 메서드 구현 → sort.Interface 사용가능
- Less() 메서드: 각 요소의 Age값 비교
- []Student를 Students 타입으로 변환 후 sort.Sort() 함수 호출
- Students(s): []Student 타입인 s를 정렬 인터페이스를 포함한 타입인 Students 타입으로 변환 ([]Studnet 타입은 Len(), Less(), Swap() 메서드 포함 X, sort.Sort() 인수로 사용 불가 → 별칭 타입을 만들어 정렬 인터페이스 포함하도록)
- 슬라이스 타입으로 만들어서 사용하다가 정렬이 필요한 경우에도 별도 타입을 만들어서 변환하는 경우가 빈번하다.
- 이 방식은 sort.Intn() 함수가 내부에서 동작하는 방식, 따라서 Age순으로 정렬됨.
- 아직 메서드와 인터페이스 다루지 않았으므로 구조체 슬라이스는 이런 식이다~만 알아도 된다.

연습문제

1. [1 2 3 100 5]
2. slice[:len(slice)-2]
3. 6 [1, 2, 3, 4, 5]
4.

```
package main

import (
  "fmt"
  "sort"
)

type Player struct {
  Name string
  Age  int
  Goal int
  Pass float64
}

type Players []Player

func (s Players) Len() int           { return len(s) }
func (s Players) Less(i, j int) bool { return s[i].Goal > s[j].Goal }
func (s Players) Swap(i, j int)      { s[i], s[j] = s[j], s[i] }
```

```
func main() {
  s := []Player{
    {"나통키", 13, 45, 78.4}, {"오맹태", 16, 24, 67.4},
    {"오동도", 18, 54, 50.8}, {"황금산", 16, 36, 89.7}}

  sort.Sort(Players(s))
  fmt.Println(s)
}
```

출력문

```
[{오동도 18 54 50.8} {나통키 13 45 78.4} {황금산 16 36 89.7} {오맹태 16 24 67.4}]
```