**Packages**

Every Go program is made up of packages.

Programs start running in package main.

This program is using the packages with import paths "fmt" and "math/rand".

By convention, the package name is the same as the last element of the import path. For instance, the "math/rand" package comprises files that begin with the statement package rand.

**Note:** The environment in which these programs are executed is deterministic, so each time you run the example program rand.Intn will return the same number.

(To see a different number, seed the number generator; see [rand.Seed](https://go.dev/pkg/math/rand/" \l "Seed" \t "_self). Time is constant in the playground, so you will need to use something else as the seed.)

## Exported names

In Go, a name is exported if it begins with a capital letter. For example, Pizza is an exported name, as is Pi, which is exported from the math package.

pizza and pi do not start with a capital letter, so they are not exported.

When importing a package, you can refer only to its exported names. Any "unexported" names are not accessible from outside the package.

Run the code. Notice the error message.

To fix the error, rename math.pi to math.Pi and try it again.

## Functions

A function can take zero or more arguments.

In this example, add takes two parameters of type int.

Notice that the type comes *after* the variable name.



Return Multiple Result from function

## Multiple results

A function can return any number of results.

The swap function returns two strings.



# Naked Return Function

A return statement without arguments returns the named return values. This is known as a **"naked"** return. used in short function package main



## Variables

The var keywords declare list of variables as in function argument lists, the type is last.



## Short variable declarations

Inside a function, the := short assignment statement can be used in place of a var declaration with implicit type.

Outside a function, every statement begins with a keyword (var, func, and so on) and so the := construct is not available.



## Basic types

Go's basic types are

bool

string

int int8 int16 int32 int64

uint uint8 uint16 uint32 uint64 uintptr

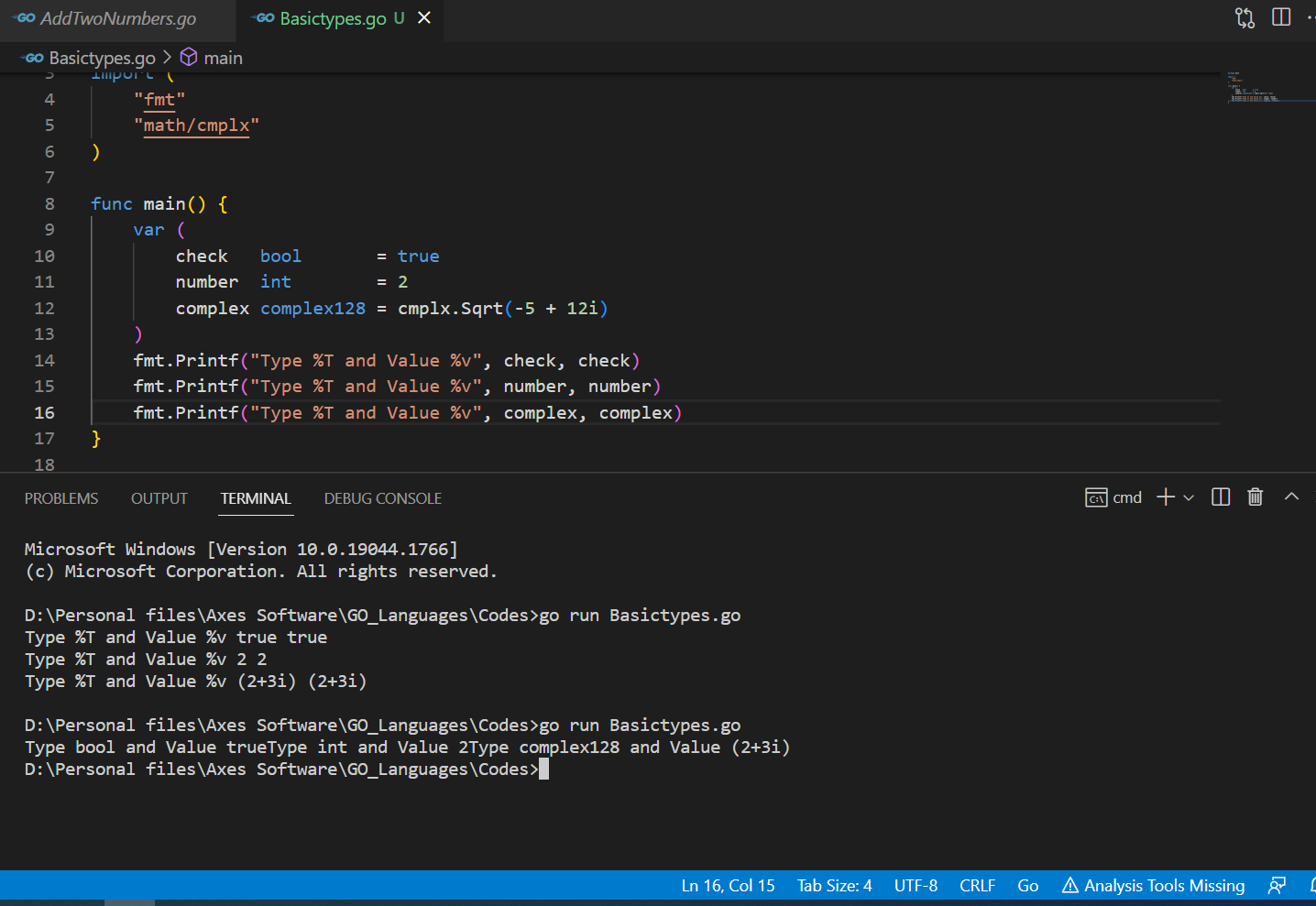
byte // alias for uint8

rune // alias for int32

// represents a Unicode code point

float32 float64

complex64 complex128



## Constants

Constants are declared like variables, but with the const keyword.

Constants can be character, string, boolean, or numeric values.

Constants cannot be declared using the := syntax.

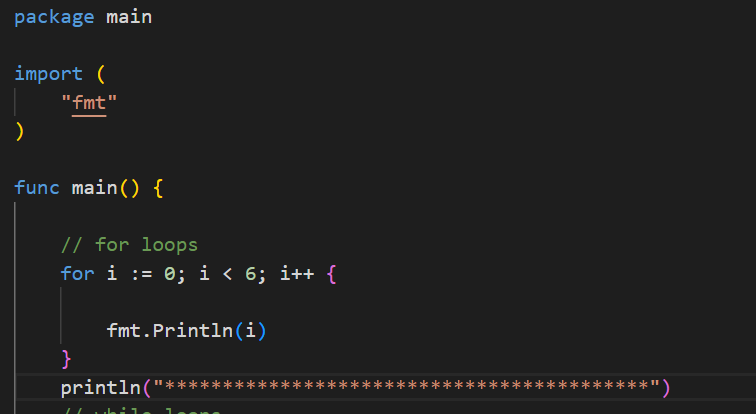
**Flow Control Statement**

## For

The basic for loop has three components separated by semicolons:

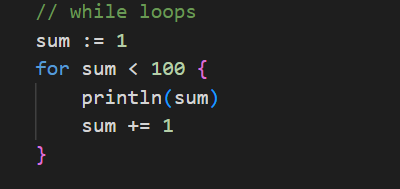
* the init statement: executed before the first iteration
* the condition expression: evaluated before every iteration
* the post statement: executed at the end of every iteration

The init statement will often be a short variable declaration, and the variables declared there are visible only in the scope of the for statement.



## For is Go's "while"

At that point you can drop the semicolons: C's while is spelled for in Go.



**Variables and types**

    // to know type - type %t

    fmt.Printf("%T \n", 10)

    // to know the value %v

    fmt.Printf("%v \n", 10)

    // to know the value of bool %t

    fmt.Printf("%t \n", true)

    // integer %b base 2

    // %o base 8

    // %x base 16

    // %d base 10

    fmt.Printf("%d \n", 10)

    fmt.Printf("%o \n", 10)

    fmt.Printf("%x \n", 10)

    fmt.Printf("%d \n", 10)

    // floating point

    // %e scientif notation

    // %f decimal

    // %g large exponent

    fmt.Printf("%e \n", 10.4)

    fmt.Printf("%f \n", 10.4)

    fmt.Printf("%g \n", 10.4)

    // string

    // %s

    // %q

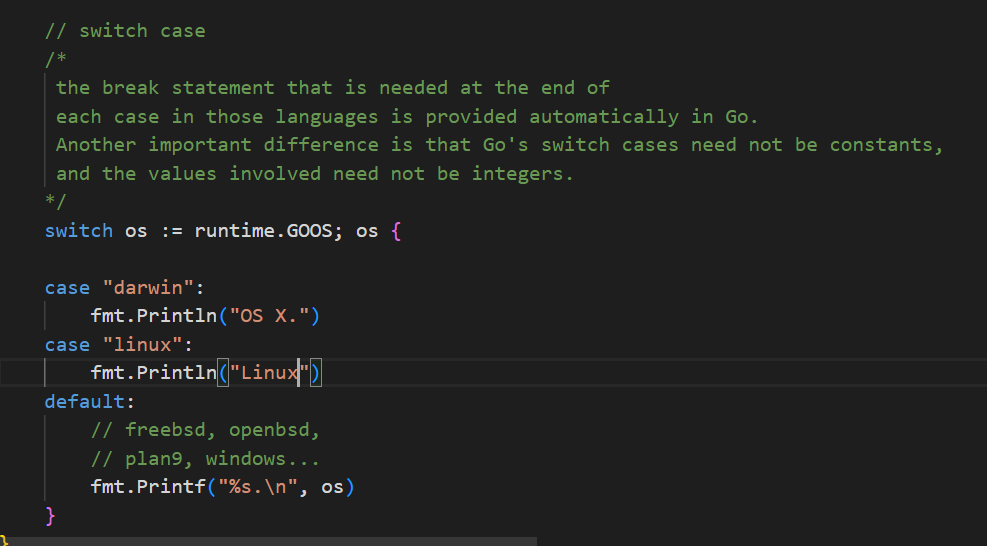
    fmt.Printf("%s \n", "Hello")

    fmt.Printf("%q \n", "Hello")

**Switch case**

A switch statement is a shorter way to write a sequence of if - else statements. It runs the first case whose value is equal to the condition expression

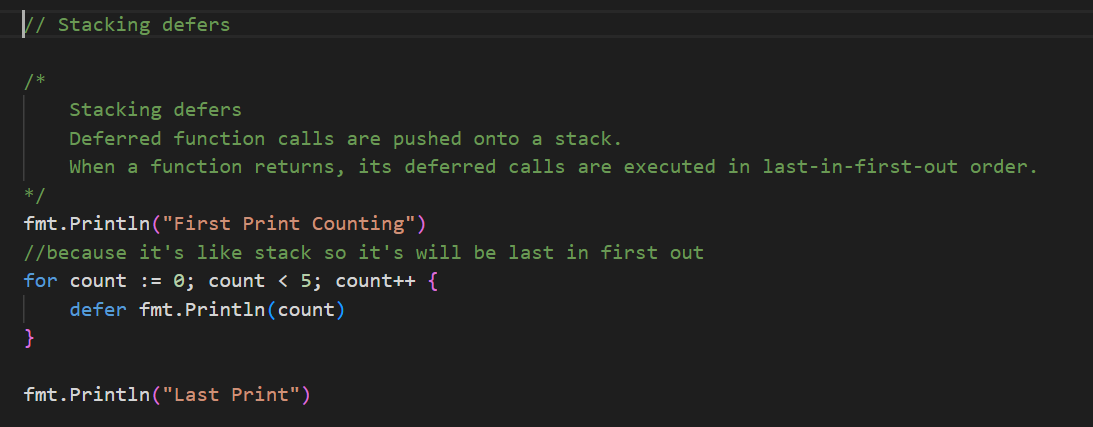
the \*\*break\*\* statement that is needed at the end of each case in those languages is provided automatically in Go. Another important difference is that Go's switch cases need not be constants, and the values involved need not be integers.



**Defer**

## Stacking defers

Deferred function calls are pushed onto a stack. When a function returns, its deferred calls are executed in last-in-first-out order.



## Pointers

Go has pointers. A pointer holds the memory address of a value.

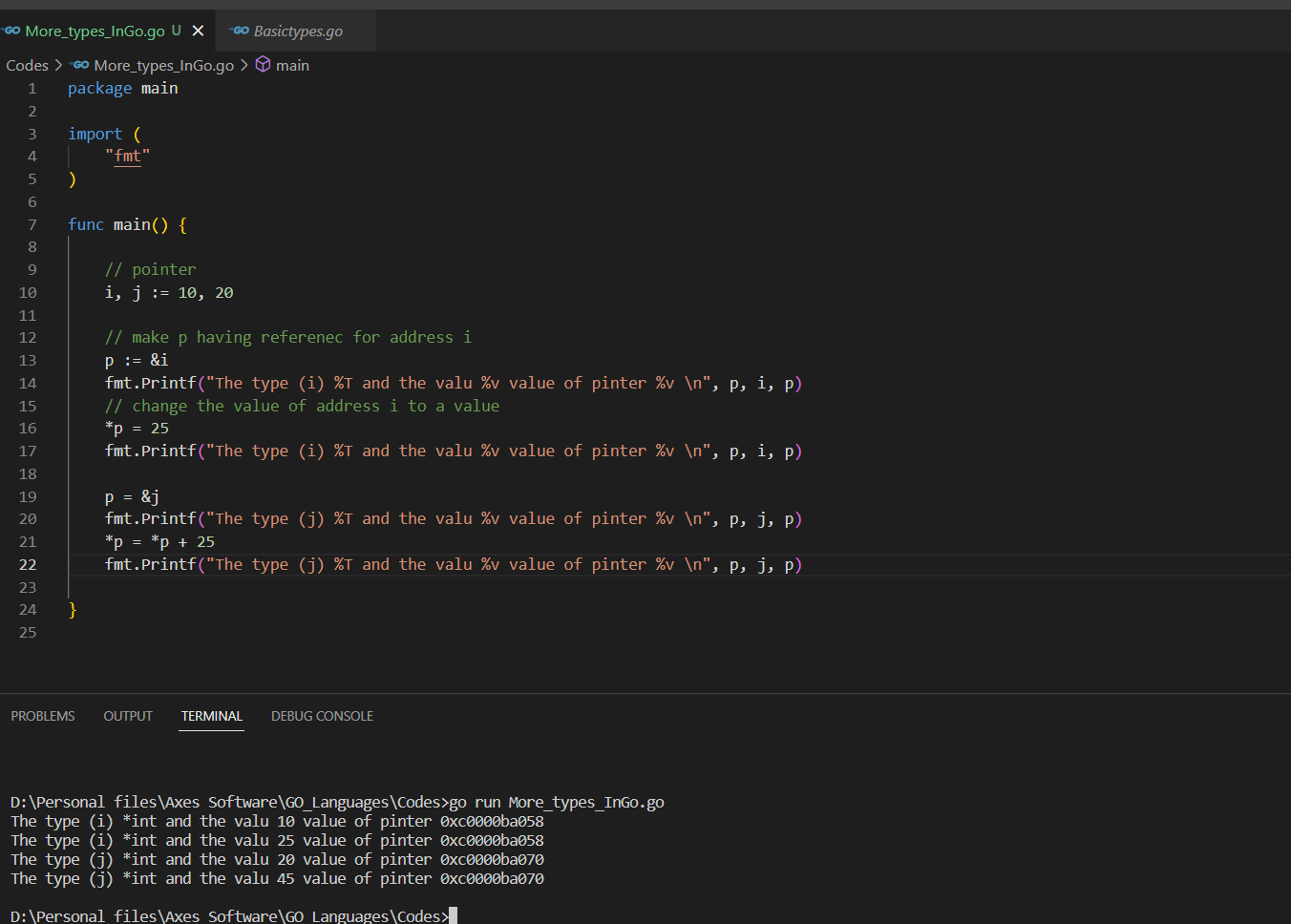
The type \*T is a pointer to a T value. Its zero value is nil.

var p \*int

The & operator generates a pointer to its operand.

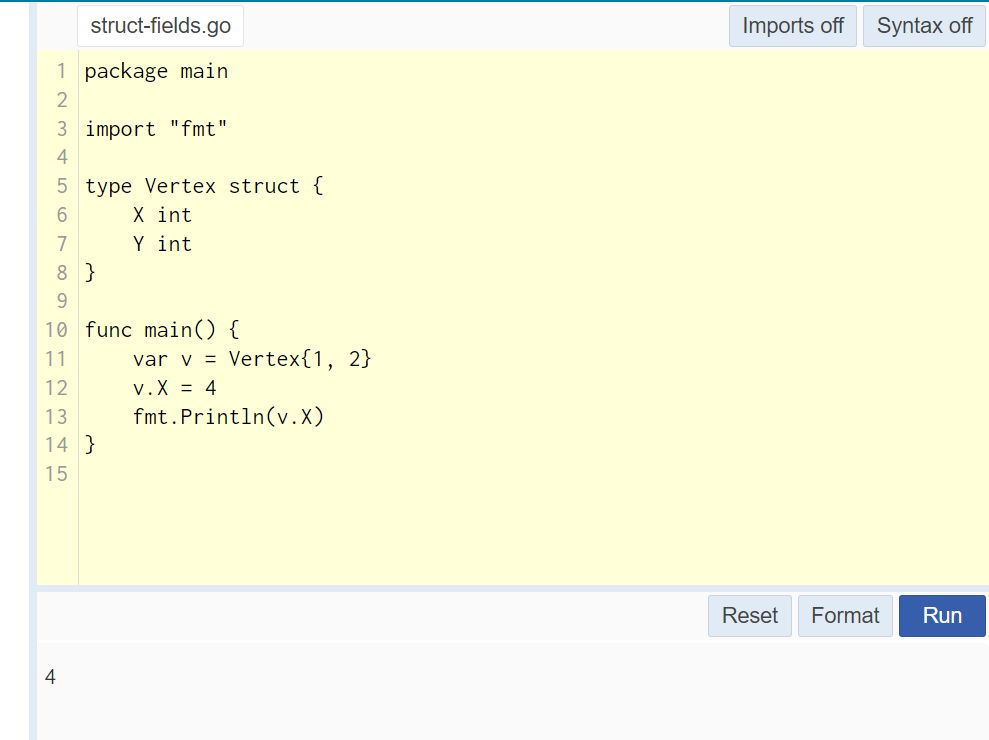
i := 42

p = &i



## Structs

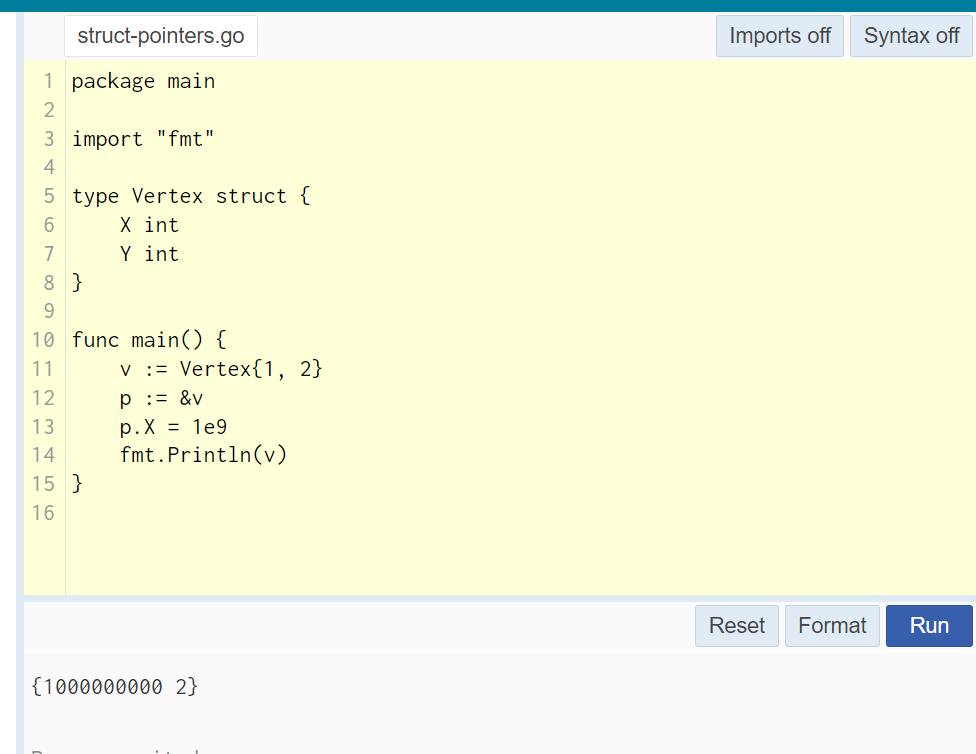
A struct is a collection of fields.



## Pointers to structs

Struct fields can be accessed through a struct pointer.

To access the field X of a struct when we have the struct pointer p we could write (\*p).X. However, that notation is cumbersome, so the language permits us instead to write just p.X, without the explicit dereference.



## Arrays

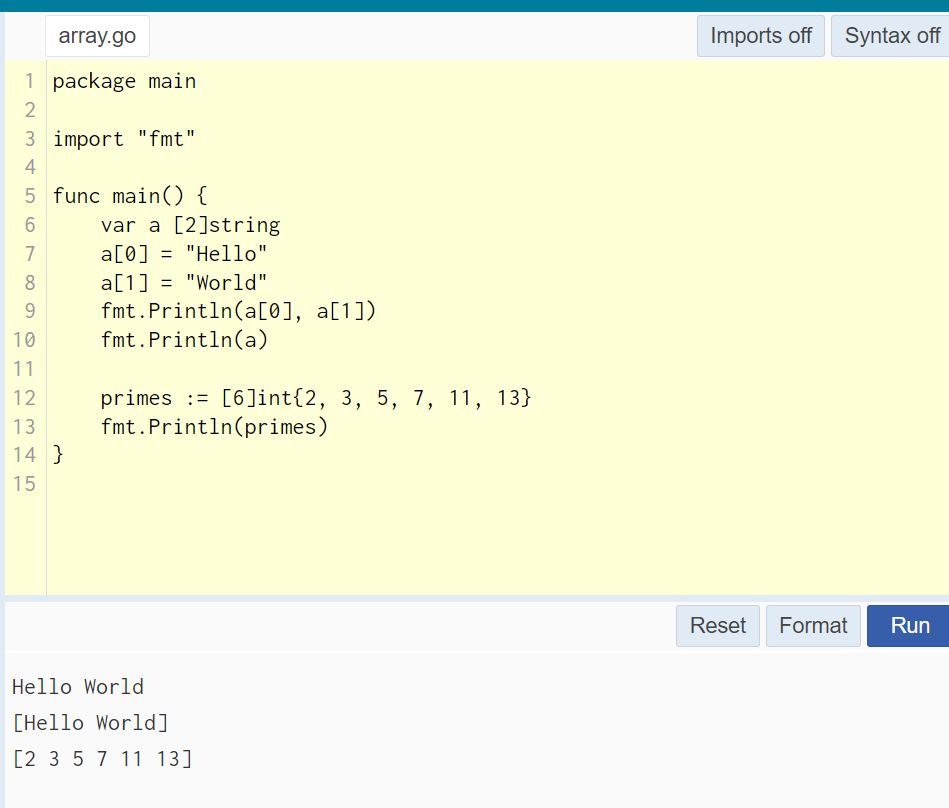
The type [n]T is an array of n values of type T.

The expression

var a [10]int

declares a variable a as an array of ten integers.

An array's length is part of its type, so arrays cannot be resized. This seems limiting, but don't worry; Go provides a convenient way of working with arrays.



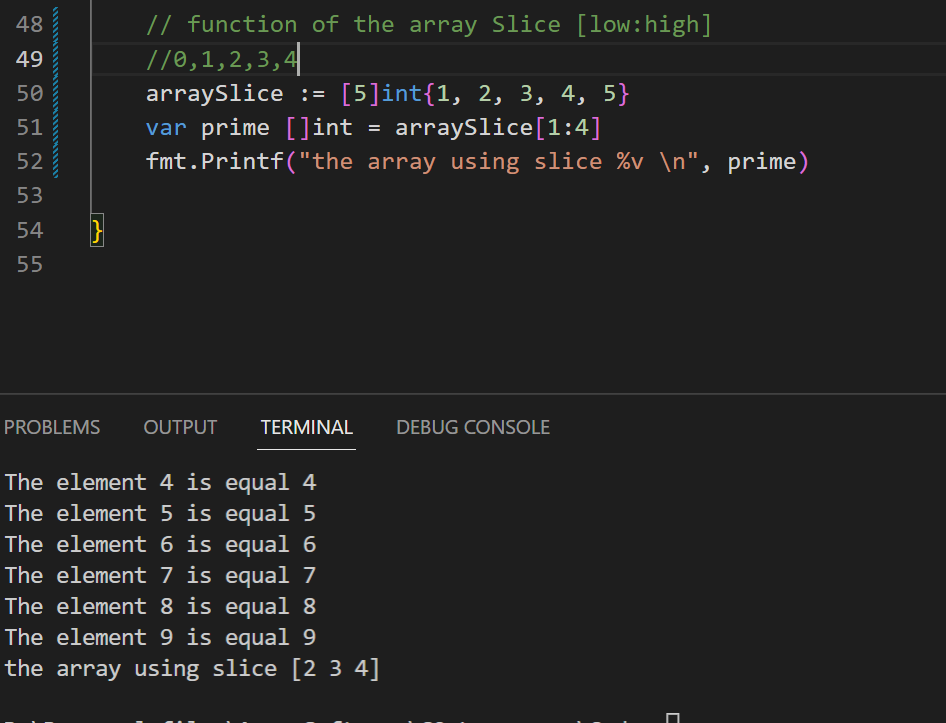
**Slice**

An array has a fixed size. A slice, on the other hand, is a dynamically-sized, flexible view into the elements of an array. In practice, slices are much more common than arrays.

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

A slice is formed by specifying two indices, a low and high bound, separated by a colon:

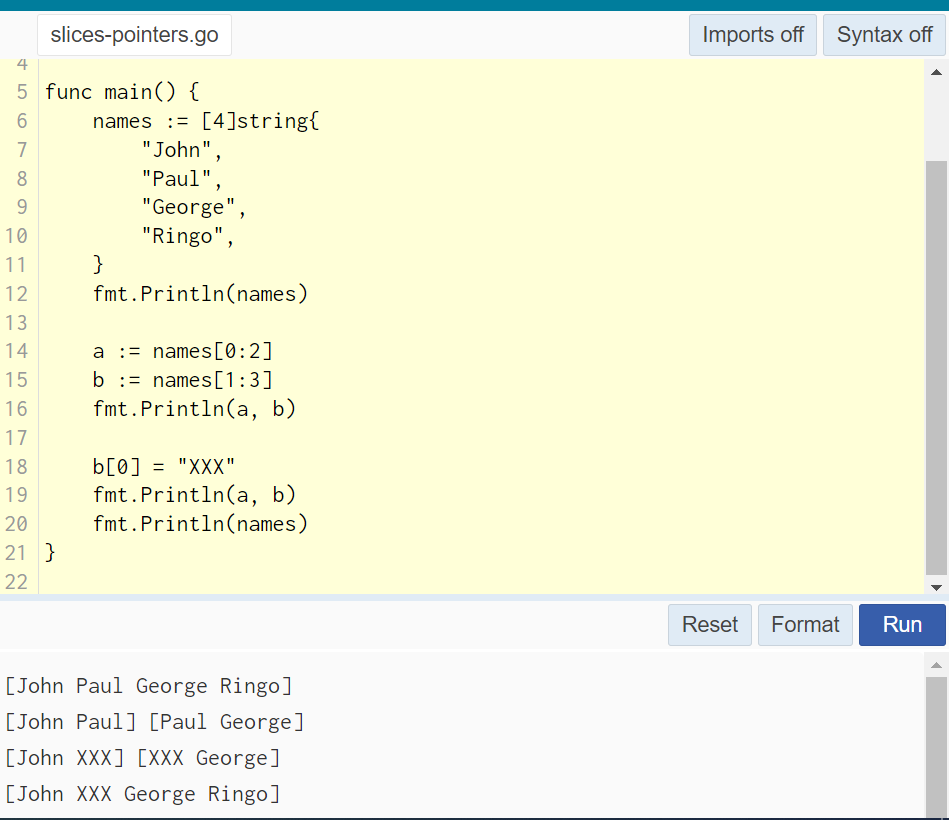
a[low : high]



## Slices are like references to arrays

A slice does not store any data, it just describes a section of an underlying array.

Changing the elements of a slice modifies the corresponding elements of its underlying array.

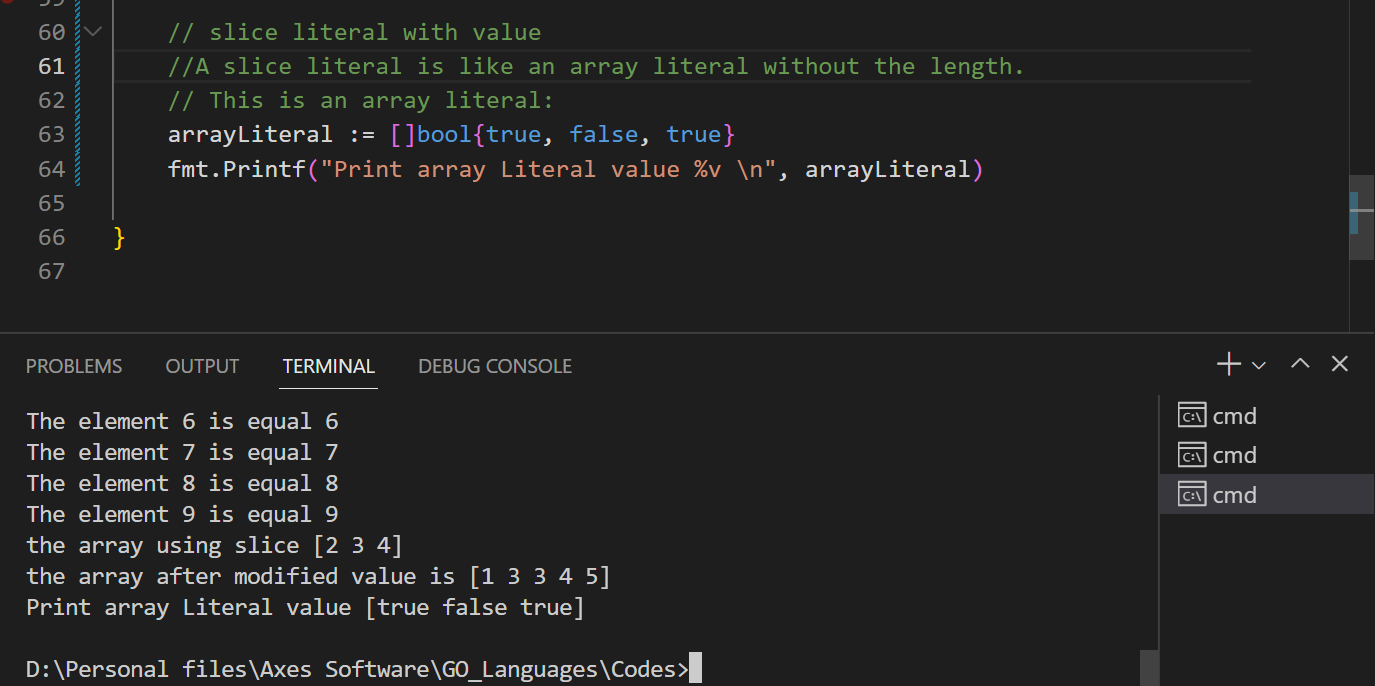


## Slice literals

A slice literal is like an array literal without the length.

This is an array literal:

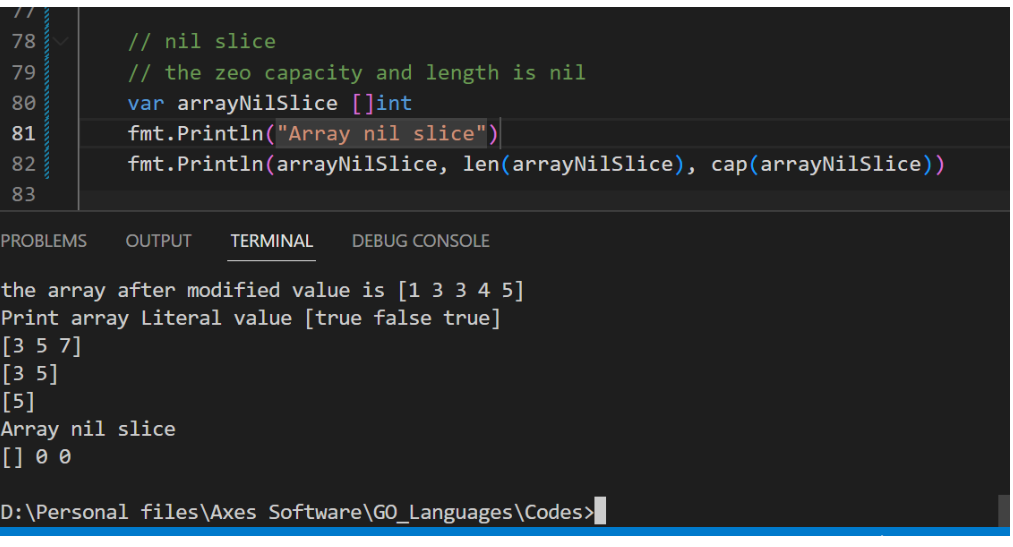
[3]bool{true, true, false}



# **Nil slices**

The zero value of a slice is nil.

A nil slice has a length and capacity of 0 and has no underlying array.



## Creating a slice with make

Slices can be created with the built-in make function; this is how you create dynamically-sized arrays.

The make function allocates a zeroed array and returns a slice that refers to that array:

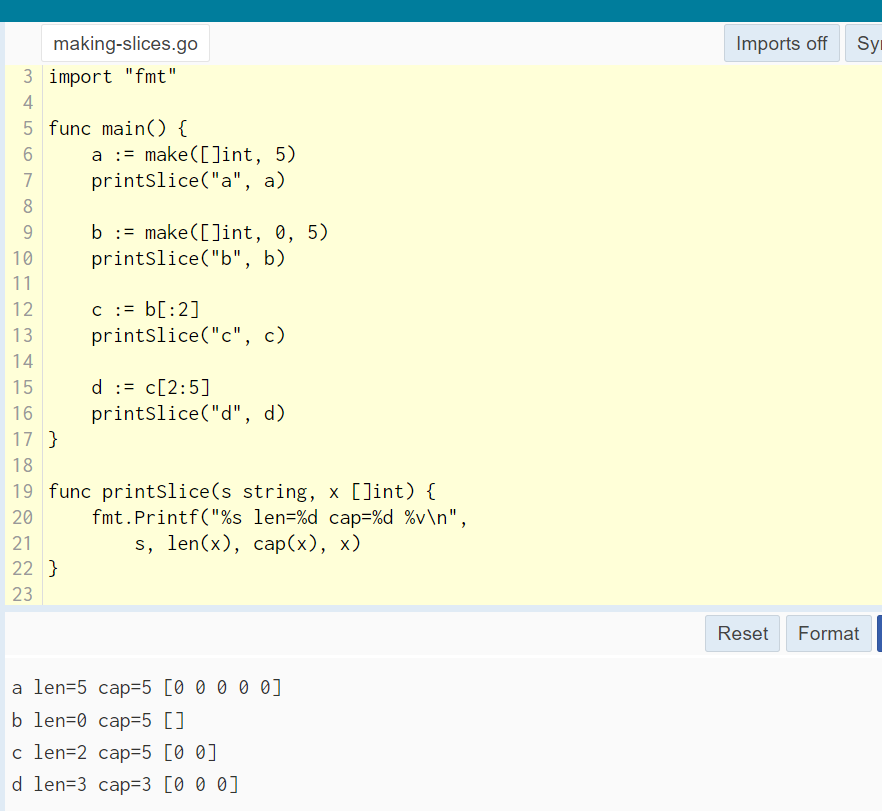
a := make([]int, 5) // len(a)=5

To specify a capacity, pass a third argument to make:

b := make([]int, 0, 5) // len(b)=0, cap(b)=5

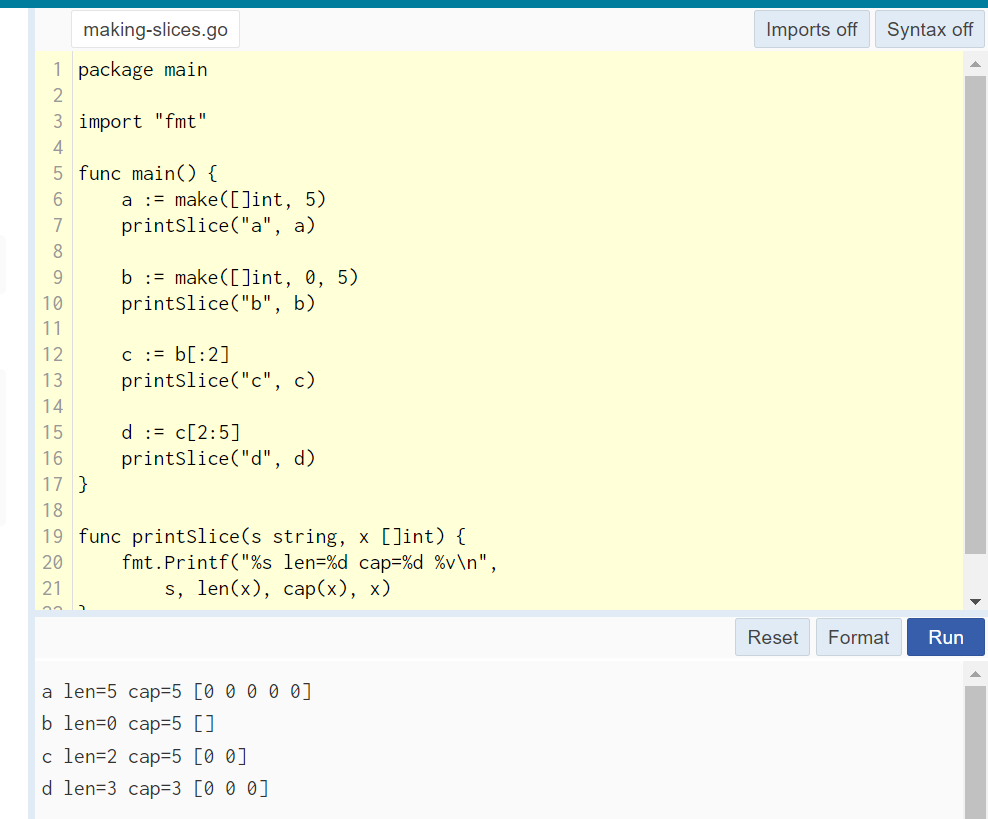
b = b[:cap(b)] // len(b)=5, cap(b)=5

b = b[1:] // len(b)=4, cap(b)=4



## Slices of slices

Slices can contain any type, including other slices.



## Appending to a slice

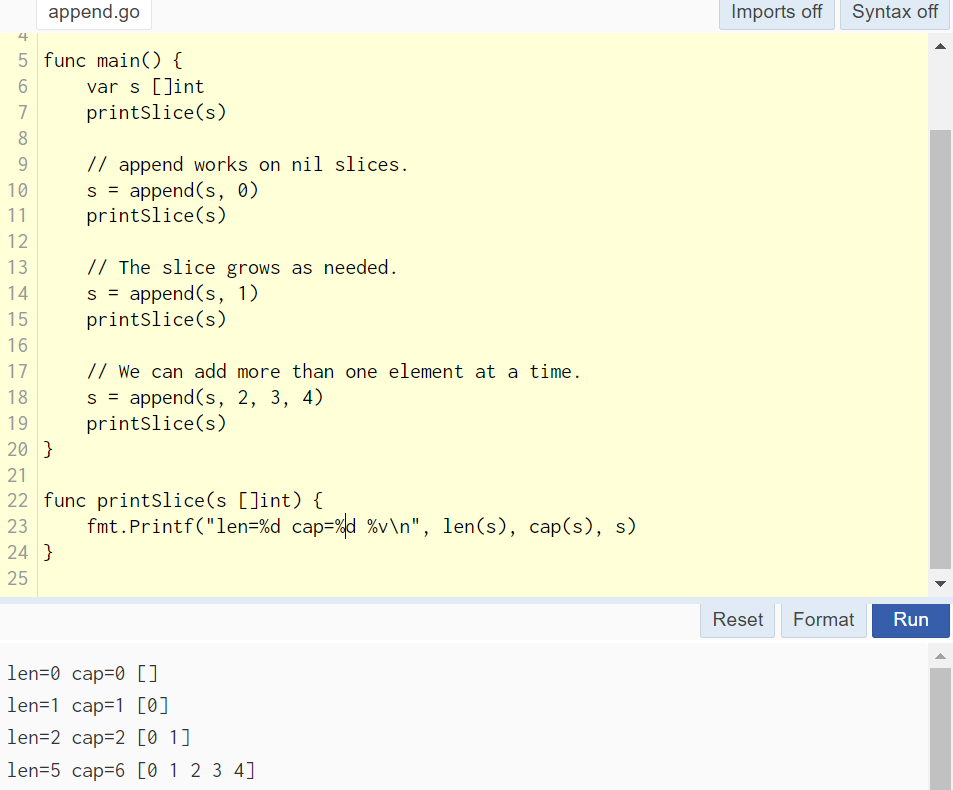
It is common to append new elements to a slice, and so Go provides a built-in append function. The [documentation](https://go.dev/pkg/builtin/#append) of the built-in package describes append.

func append(s []T, vs ...T) []T

The first parameter s of append is a slice of type T, and the rest are T values to append to the slice.

The resulting value of append is a slice containing all the elements of the original slice plus the provided values.

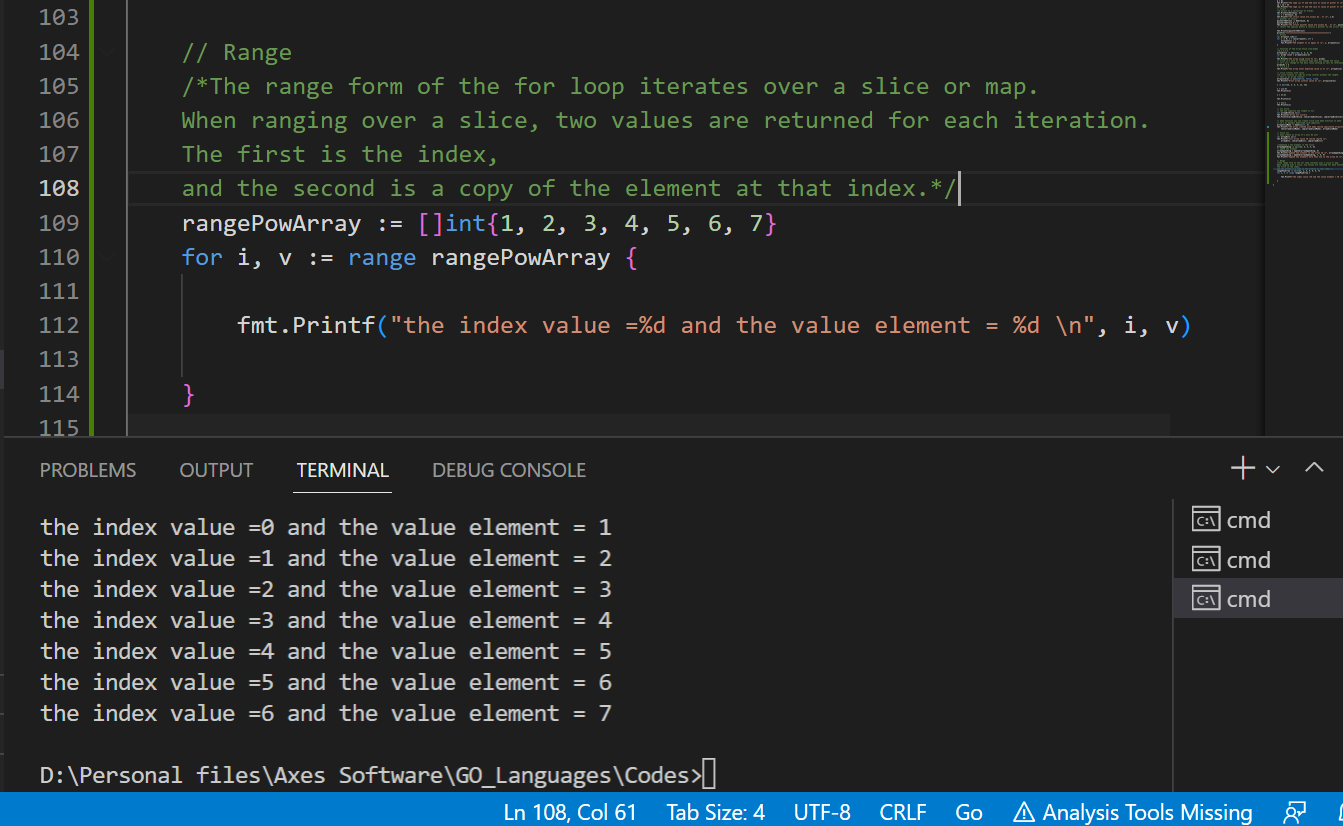
If the backing array of s is too small to fit all the given values a bigger array will be allocated. The returned slice will point to the newly allocated array.



## Range

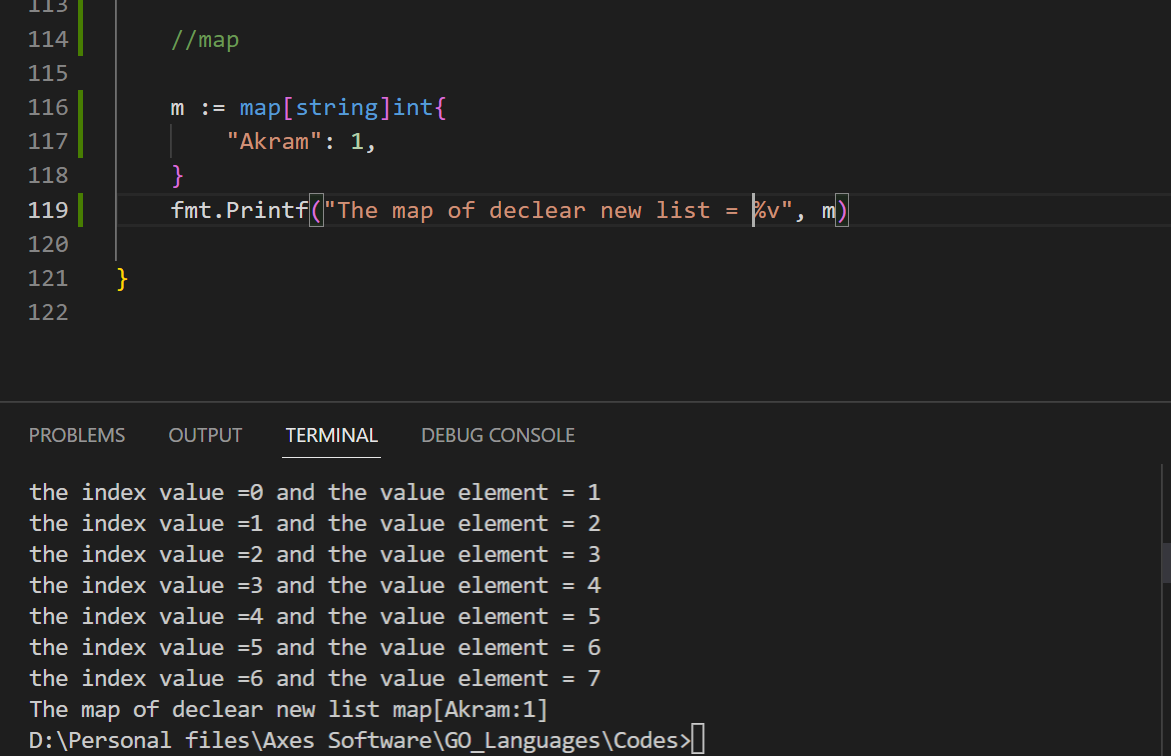
The range form of the for loop iterates over a slice or map.

When ranging over a slice, two values are returned for each iteration. The first is the index, and the second is a copy of the element at that index.

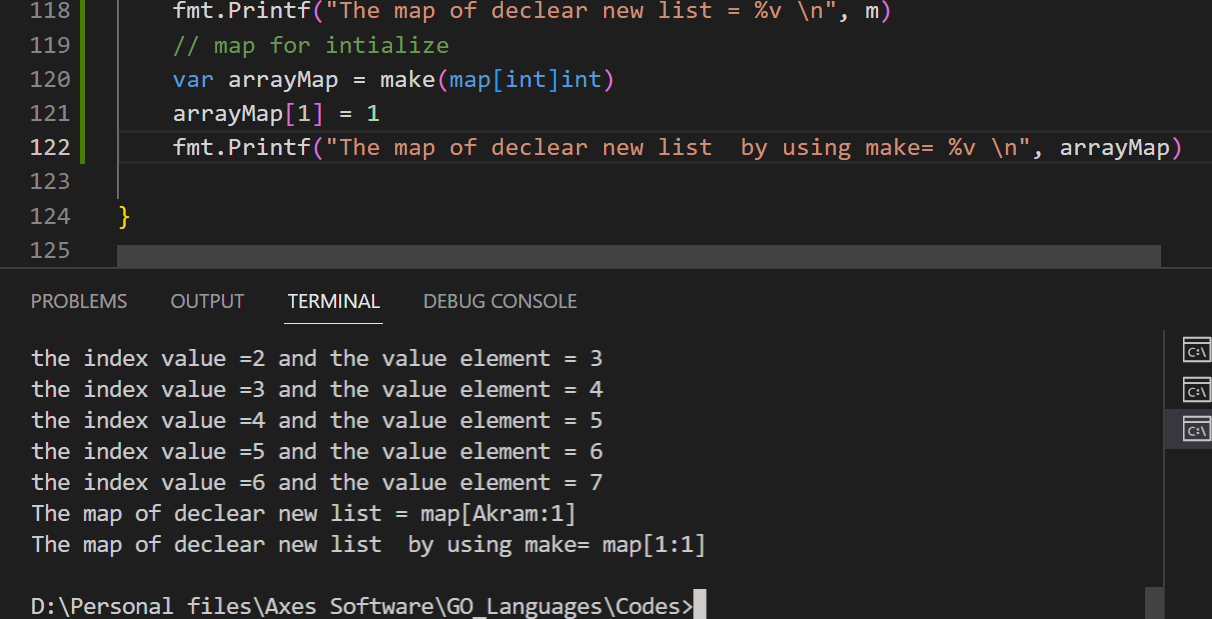


## Map literals

Map literals are like struct literals, but the keys are required.



**You can create map by using make function**



## Mutating Maps Insert\_update\_Delete

Insert or update an element in map m:

m[key] = elem

Retrieve an element:

elem = m[key]

Delete an element:

delete(m, key)

Test that a key is present with a two-value assignment:

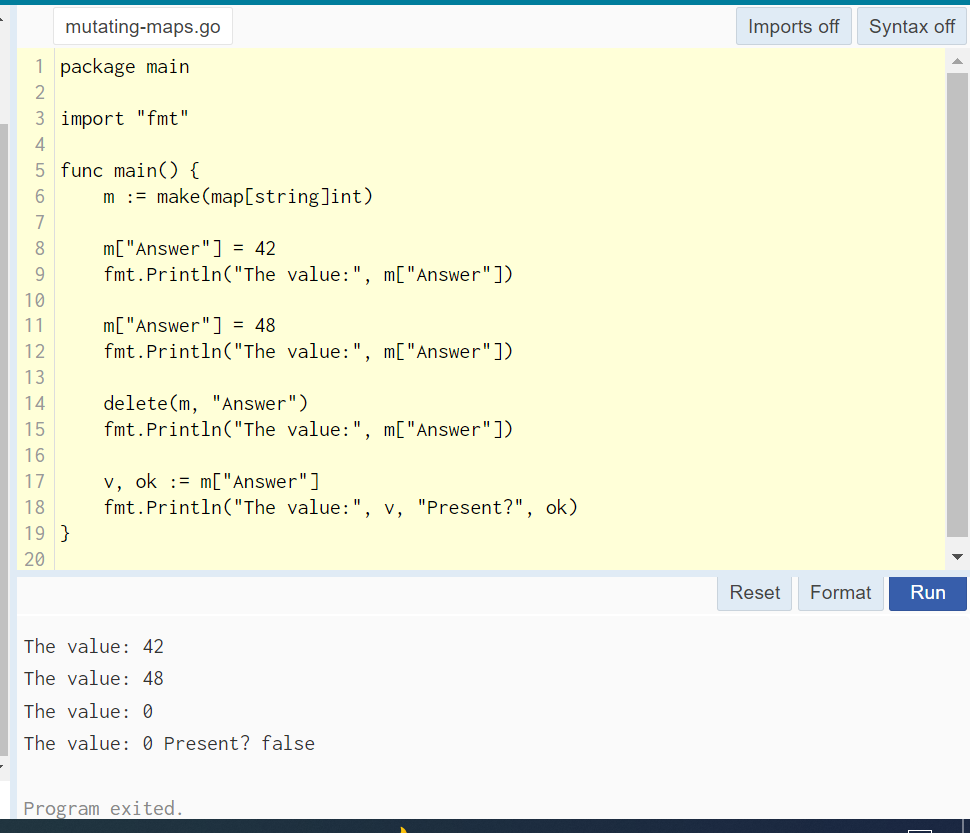
elem, ok = m[key]

If key is in m, ok is true. If not, ok is false.

If key is not in the map, then elem is the zero value for the map's element type.

**Note:** If elem or ok have not yet been declared you could use a short declaration form:

elem, ok := m[key]



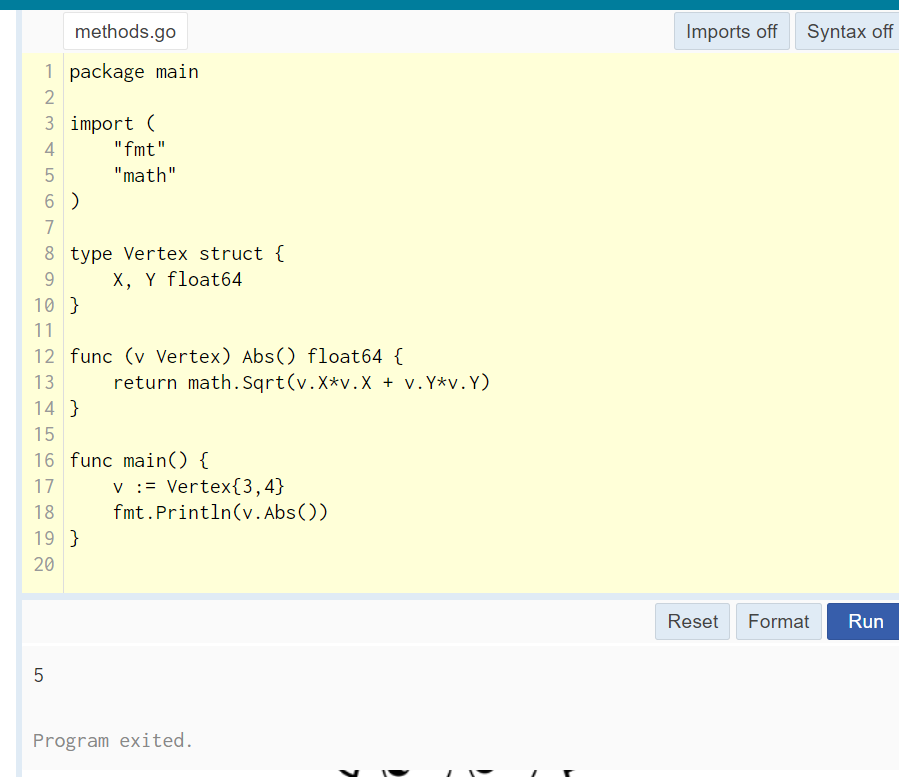
## Methods

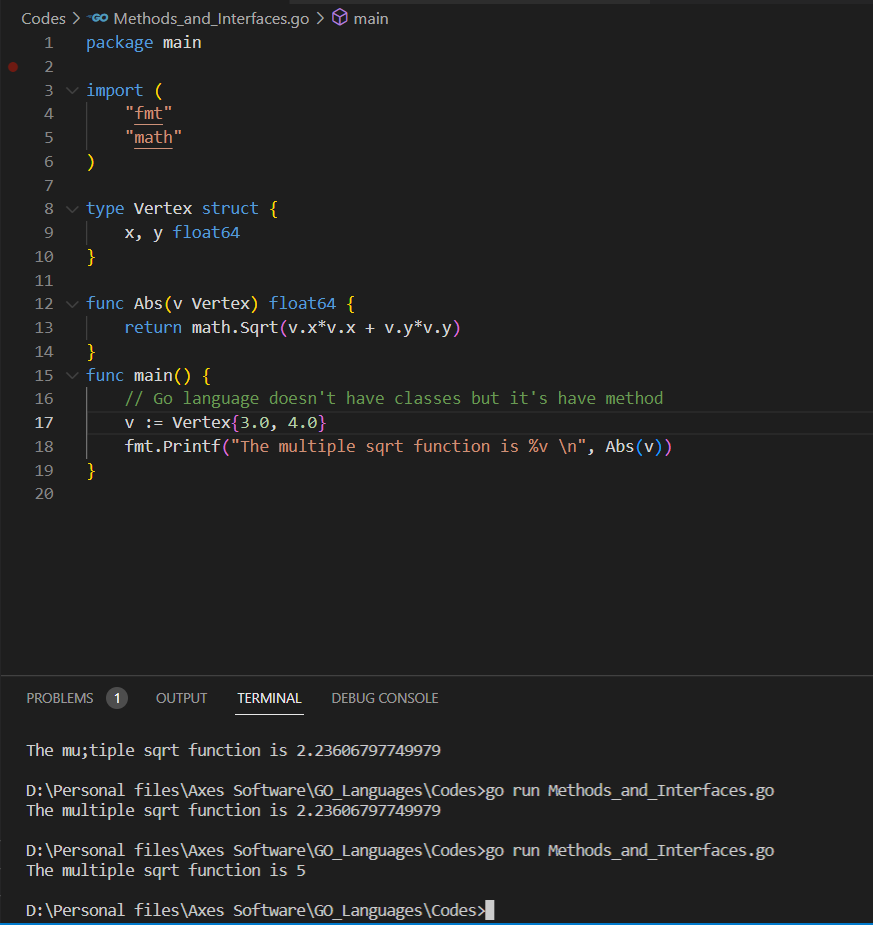
Go does not have classes. However, you can define methods on types.

A method is a function with a special *receiver* argument.

The receiver appears in its own argument list between the func keyword and the method name.

In this example, the Abs method has a receiver of type Vertex named v.



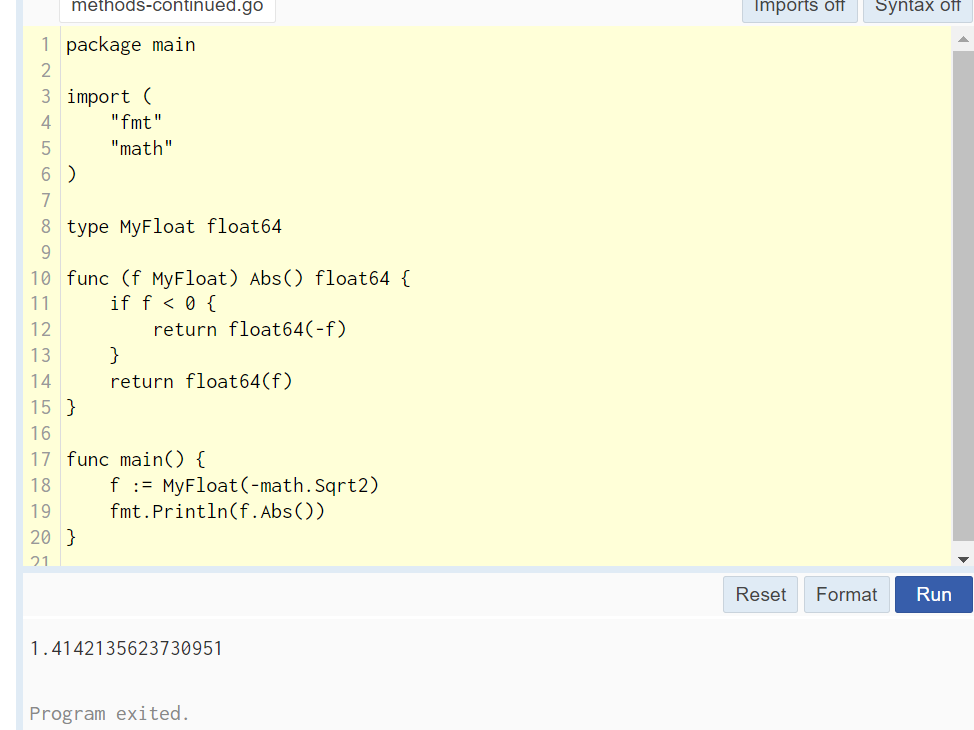


## Methods continued

You can declare a method on non-struct types, too.

In this example we see a numeric type MyFloat with an Abs method.

You can only declare a method with a receiver whose type is defined in the same package as the method. You cannot declare a method with a receiver whose type is defined in another package (which includes the built-in types such as int)



## Pointer receivers

You can declare methods with pointer receivers.

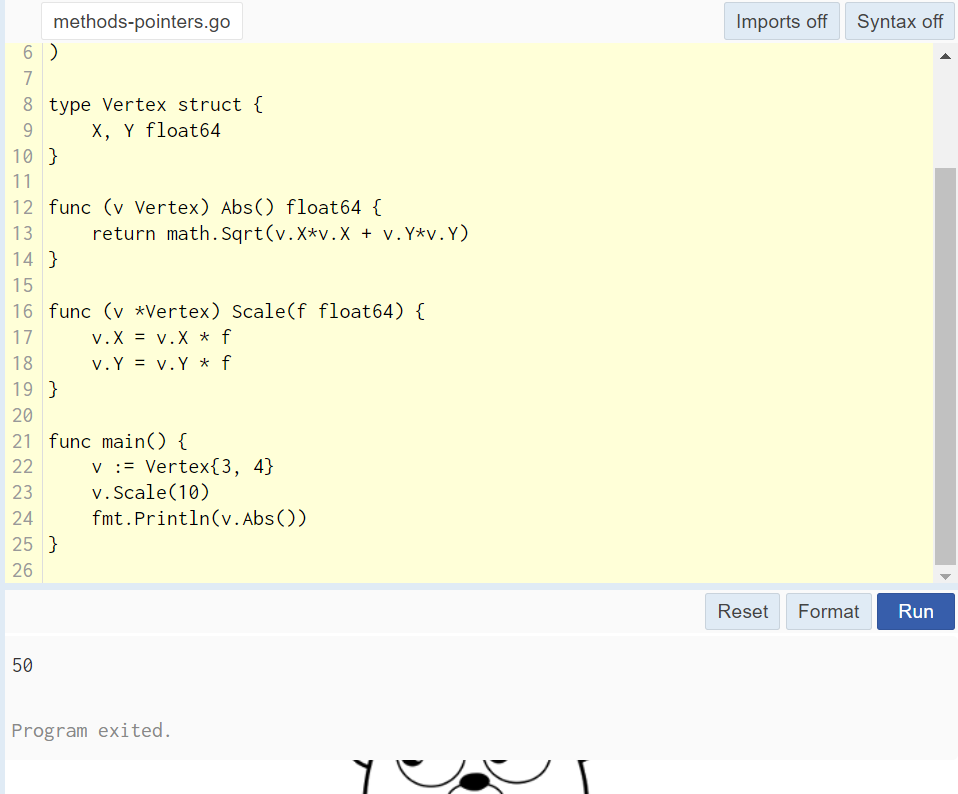
This means the receiver type has the literal syntax \*T for some type T. (Also, T cannot itself be a pointer such as \*int.)

For example, the Scale method here is defined on \*Vertex.

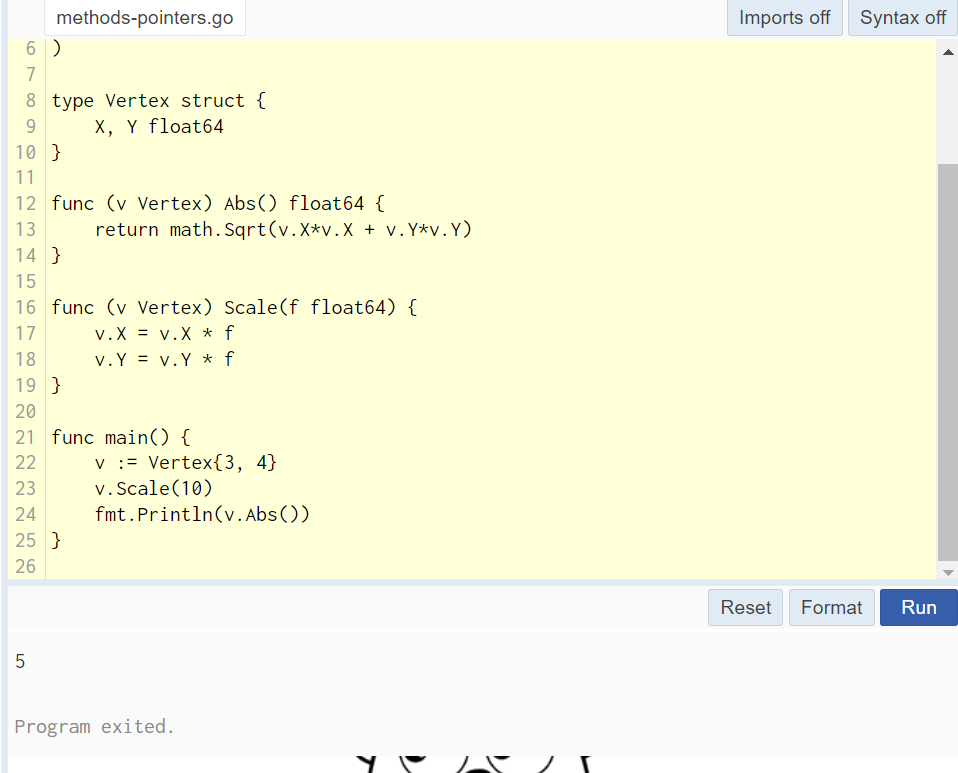
Methods with pointer receivers can modify the value to which the receiver points (as Scale does here). Since methods often need to modify their receiver, pointer receivers are more common than value receivers.

Try removing the \* from the declaration of the Scale function on line 16 and observe how the program's behavior changes.

With a value receiver, the Scale method operates on a copy of the original Vertex value. (This is the same behavior as for any other function argument.) The Scale method must have a pointer receiver to change the Vertex value declared in the main function

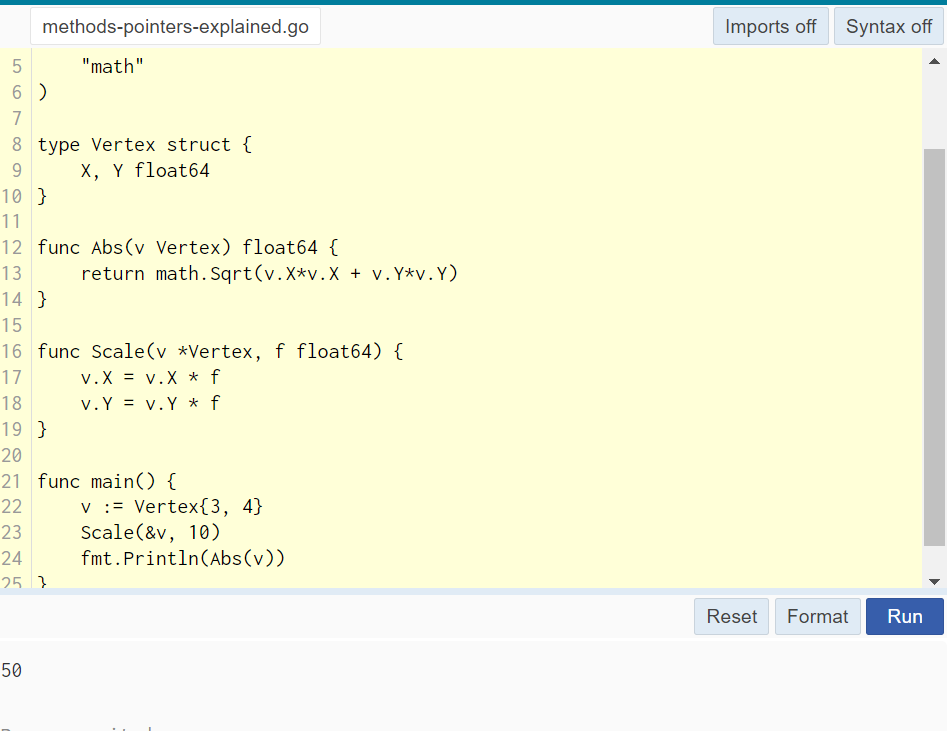


**When remove \* from func scale it’s will be**



**Another way to create functions**

**Method as pointer**



## Methods and pointer indirection

Comparing the previous two programs, you might notice that functions with a pointer argument must take a pointer:

var v Vertex

ScaleFunc(v, 5) // Compile error!

ScaleFunc(&v, 5) // OK

while methods with pointer receivers take either a value or a pointer as the receiver when they are called:

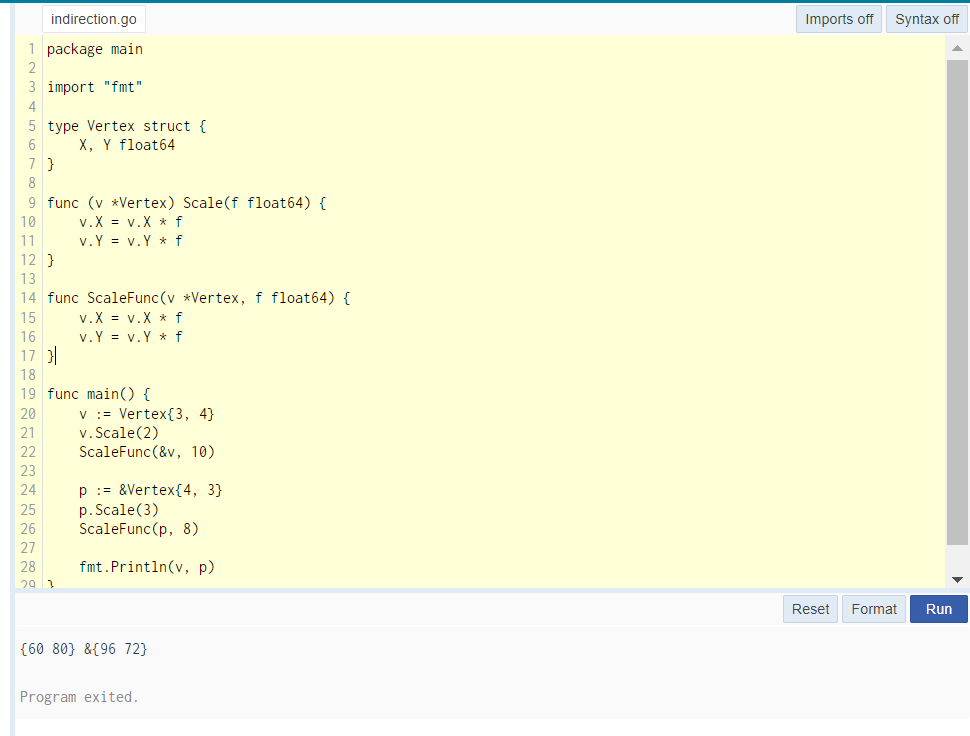
var v Vertex

v.Scale(5) // OK

p := &v

p.Scale(10) // OK

For the statement v.Scale(5), even though v is a value and not a pointer, the method with the pointer receiver is called automatically. That is, as a convenience, Go interprets the statement v.Scale(5) as (&v).Scale(5) since the Scale method has a pointer receiver.



## Methods and pointer indirection (2)

The equivalent thing happens in the reverse direction.

Functions that take a value argument must take a value of that specific type:

var v Vertex

fmt.Println(AbsFunc(v)) // OK

fmt.Println(AbsFunc(&v)) // Compile error!

while methods with value receivers take either a value or a pointer as the receiver when they are called:

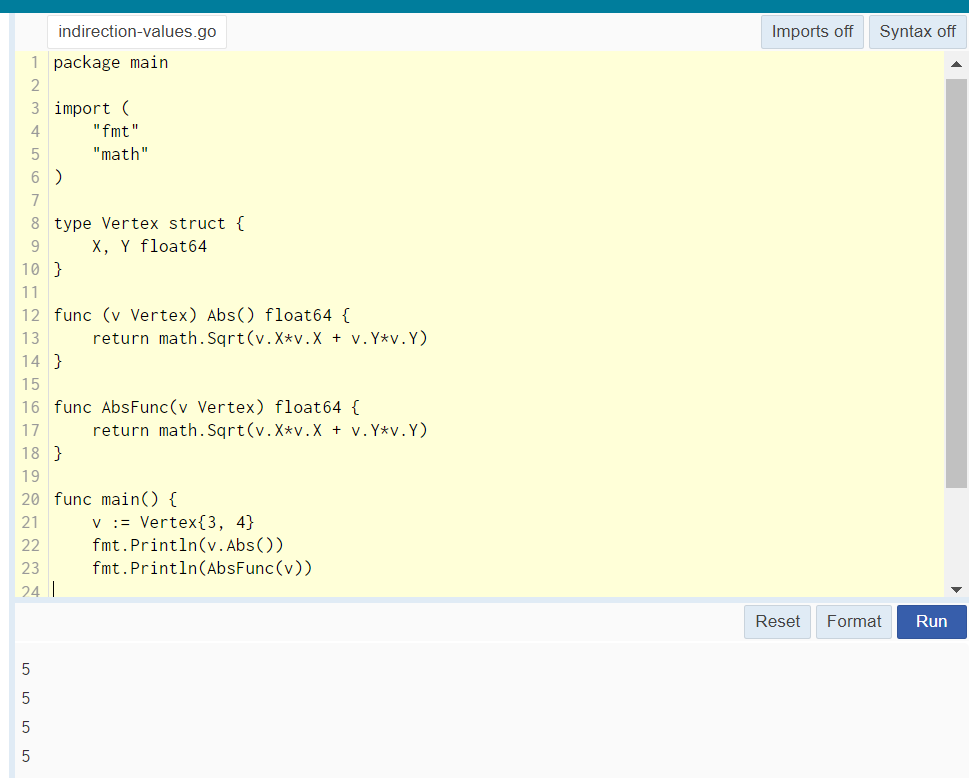
var v Vertex

fmt.Println(v.Abs()) // OK

p := &v

fmt.Println(p.Abs()) // OK

In this case, the method call p.Abs() is interpreted as (\*p).Abs().



## Choosing a value or pointer receiver

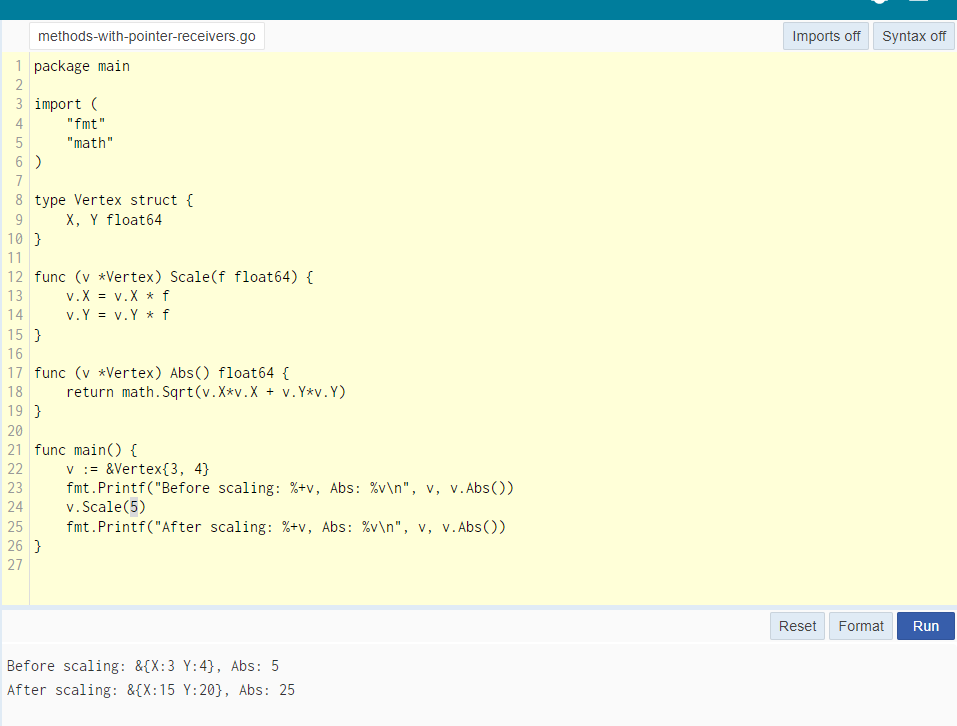
There are two reasons to use a pointer receiver.

The first is so that the method can modify the value that its receiver points to.

The second is to avoid copying the value on each method call. This can be more efficient if the receiver is a large struct, for example.

In this example, both Scale and Abs are with receiver type \*Vertex, even though the Abs method needn't modify its receiver.

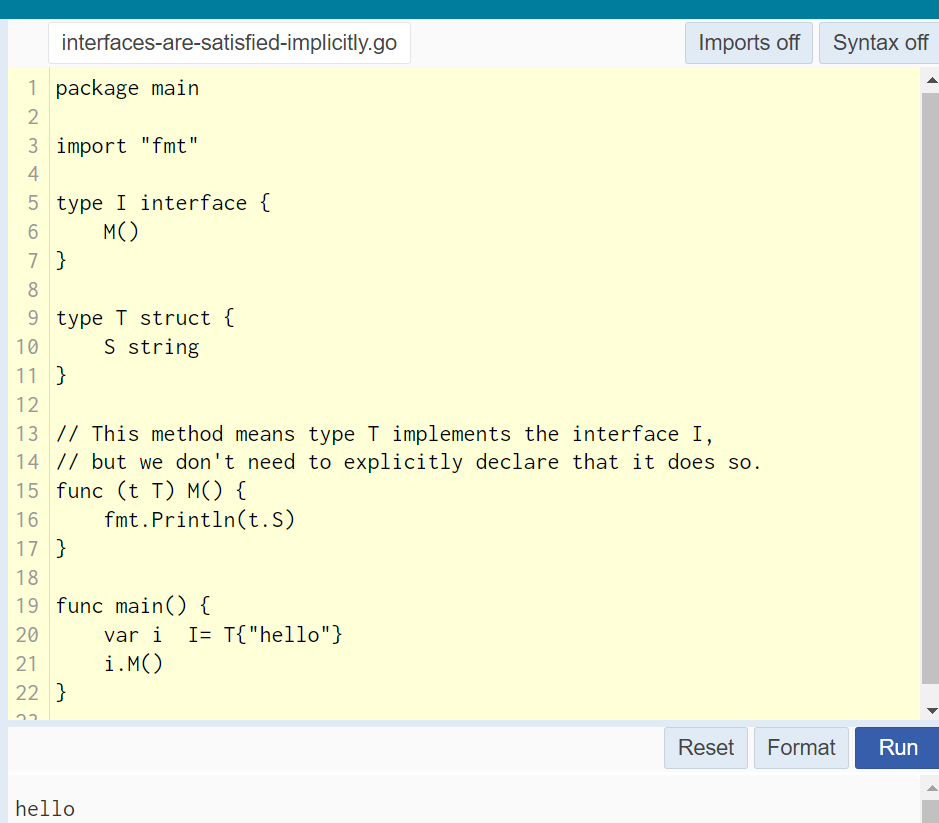
In general, all methods on a given type should have either value or pointer receivers, but not a mixture of both. (We'll see why over the next few pages.)



## Interfaces are implemented implicitly

A type implements an interface by implementing its methods. There is no explicit declaration of intent, no "implements" keyword.

Implicit interfaces decouple the definition of an interface from its implementation, which could then appear in any package without prearrangement.



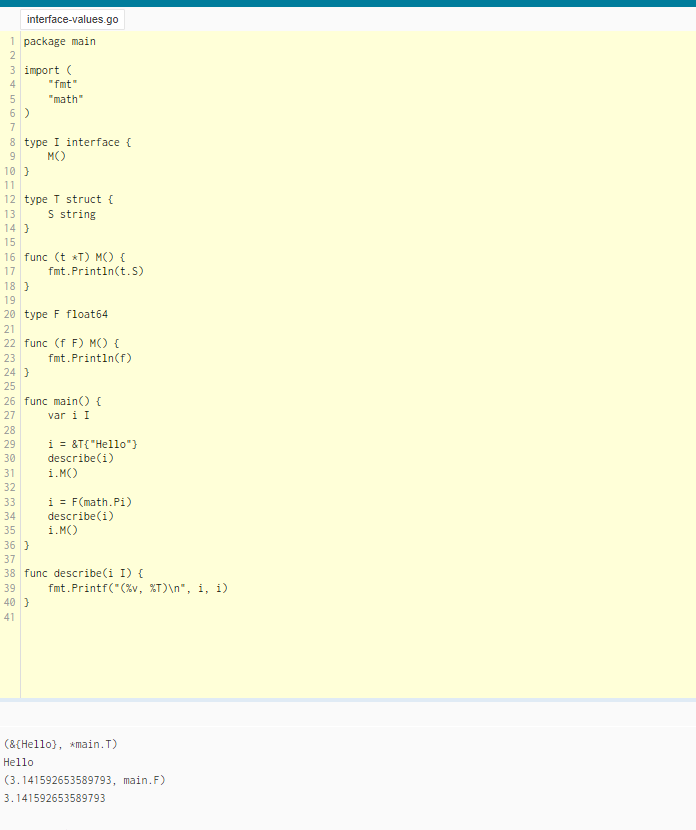
## Interface values

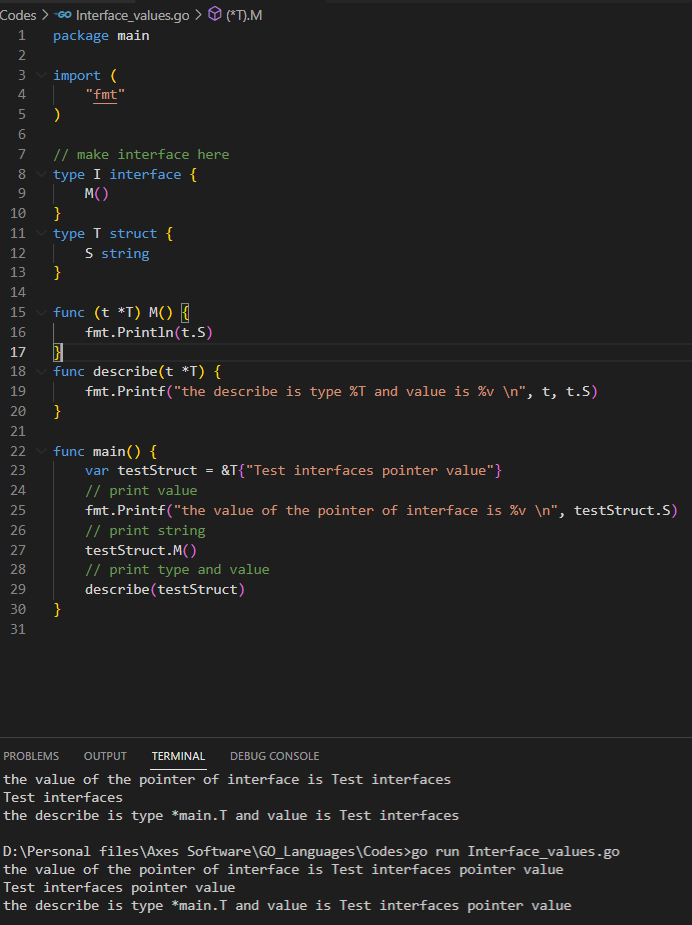
Under the hood, interface values can be thought of as a tuple of a value and a concrete type:

(value, type)

An interface value holds a value of a specific underlying concrete type.

Calling a method on an interface value executes the method of the same name on its underlying type.





# Interface values with nil underlying values

If the concrete value inside the interface itself is nil, the method will be called with a nil receiver.

In some languages this would trigger a null pointer exception, but in Go it is common to write methods that gracefully handle being called with a nil receiver (as with the method M in this example.)

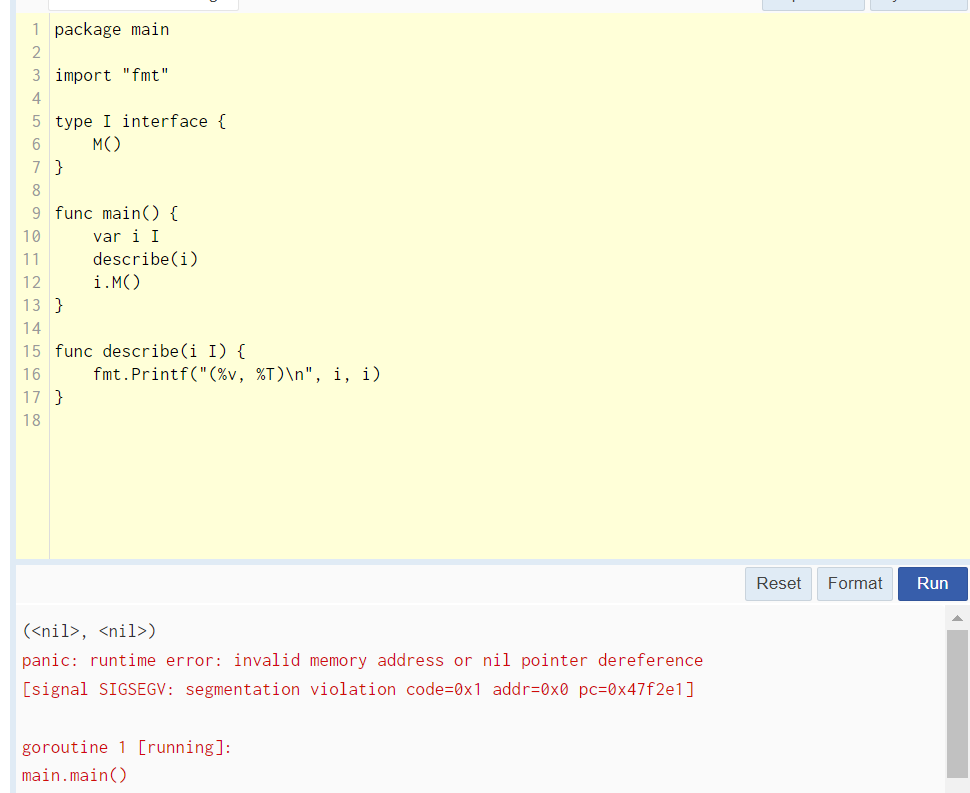
Note that an interface value that holds a nil concrete value is itself non-nil.

## 

## Nil interface values

A nil interface value holds neither value nor concrete type.

Calling a method on a nil interface is a run-time error because there is no type inside the interface tuple to indicate which *concrete* method to call.



## 

## The empty interface

The interface type that specifies zero methods is known as the *empty interface*:

interface{}

An empty interface may hold values of any type. (Every type implements at least zero methods.)

Empty interfaces are used by code that handles values of unknown type. For example, fmt.Print takes any number of arguments of type interface{}.

