Flow control statements: for, if, else, switch and defer

Learn how to control the flow of your code with conditionals, loops, switches and defers.

The Go Authors

https://golang.org

*** For**

Go has only one looping construct, the `for` loop.

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.

The loop will stop iterating once the boolean condition evaluates to `false`.

***Note:* Unlike other languages like C, Java, or JavaScript there are no parentheses**
surrounding the three components of the `for` statement and the braces `{`}` are

always required.

*** For continued**

The init and post statements are optional.

*** For is Go's "while"**

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

*** Forever**

If you omit the loop condition it loops forever, so an infinite loop is compactly expressed.

*** If**

Go's `if` statements are like its `for` loops; the expression need not be

surrounded by parentheses `(`)` but the braces `{`}` are required.

*** If with a short statement**

Like `for`, the `if` statement can start with a short statement to execute before the condition.

Variables declared by the statement are only in scope until the end of the `if`.

(Try using `v` in the last `return` statement.)

*** If and else**

Variables declared inside an `if` short statement are also available inside any

of the `else` blocks.

(Both calls to `pow` return their results before the call to `fmt.Println`

in `main` begins.)

*** Exercise: Loops and Functions**

As a way to play with functions and loops, let's implement a square root function: given a number x, we want to find the number z for which z^2 is most nearly x.

Computers typically compute the square root of x using a loop.

Starting with some guess z, we can adjust z based on how close z^2 is to x,

producing a better guess:

```
 z -= (z*z - x) / (2*z)
```

Repeating this adjustment makes the guess better and better

until we reach an answer that is as close to the actual square root as can be.

Implement this in the `func`Sqrt` provided.

A decent starting guess for z is 1, no matter what the input.

To begin with, repeat the calculation 10 times and print each z along the way.

See how close you get to the answer for various values of x (1, 2, 3, ...)

and how quickly the guess improves.

Hint: To declare and initialize a floating point value,

give it floating point syntax or use a conversion:

z := 1.0

z := float64(1)

Next, change the loop condition to stop once the value has stopped

changing (or only changes by a very small amount).

See if that's more or fewer than 10 iterations.

Try other initial guesses for z, like x, or x/2.

How close are your function's results to the [[/pkg/math/#Sqrt][math.Sqrt]] in the standard library?

(*Note:* If you are interested in the details of the algorithm, the z^2 . x above

is how far away z^2 is from where it needs to be (x), and the division by 2z is the derivative
of z^2, to scale how much we adjust z by how quickly z^2 is changing.

This general approach is called
[[https://en.wikipedia.org/wiki/Newton%27s_method][Newton's method]].
It works well for many functions but especially well for square root.)

*** Switch**

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.

Go's switch is like the one in C, C++, Java, JavaScript, and PHP, except that Go only runs the selected case, not all the cases that follow.

In effect, 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.

*** Switch evaluation order**

Switch cases evaluate cases from top to bottom, stopping when a case succeeds.

(For example,

```
 switch i {

 case 0:

 case f():

 }
```

does not call `f` if `i==0`.)

#appengine: *Note:* Time in the Go playground always appears to start at

#appengine: 2009-11-10 23:00:00 UTC, a value whose significance is left as an

#appengine: exercise for the reader.

*** Switch with no condition**

Switch without a condition is the same as `switch`true`.

This construct can be a clean way to write long if-then-else chains.

*** Defer**

A defer statement defers the execution of a function until the surrounding function returns.

The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.

*** 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.

To learn more about defer statements read this

[[/blog/defer-panic-and-recover][blog post]].

*** Congratulations!**

You finished this lesson!

You can go back to the list of [[/tour/list][modules]] to find what to learn next, or continue with the [[javascript:click('.next-page')][next lesson]].