Get Programming with Go

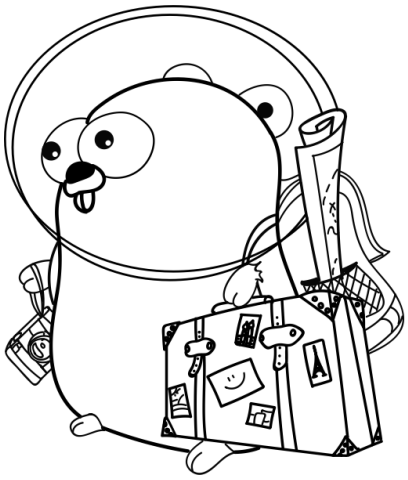
Nathan Youngman

Version v0.14.0.pre.2

Capstone 1. Ticket to Mars

Welcome to the first challenge. It's time to take everything covered in unit 1 and write a program on your own.

When planning a trip to Mars, it would be handy to have ticket pricing from multiple spacelines in one place. Websites exist that aggregate ticket prices for airlines, but so far nothing exists for spacelines. That's not a problem for you though. You can use Go to teach your computer to solve problems like this.



Start by building a prototype that generates ten random tickets and displays them in a tabular format with a nice header as follows:

```
Spaceline        Days Round-trip Price
=====================================
Virgin Galactic    23 Round-trip $  96
Virgin Galactic    39 One-way    $  37
SpaceX             31 One-way    $  41
Space Adventures   22 Round-trip $ 100
Space Adventures   22 One-way    $  50
Virgin Galactic    30 Round-trip $  84
Virgin Galactic    24 Round-trip $  94
Space Adventures   27 One-way    $  44
Space Adventures   28 Round-trip $  86
SpaceX             41 Round-trip $  72
```

The table should have four columns:

- the spaceline providing the service
- the duration for the trip to Mars (one-way)
- whether or not the price covers a return trip
- the price in millions ☹

For each ticket, randomly select one of the following spacelines: Space Adventures, SpaceX, or Virgin

Galactic.

Use a July 27, 2018 departure date for all tickets. Mars will be 57,600,000 km away from the Earth at this time.

If you are reading this after July 27, 2018, use the next suitable departure date. Mars will be 62,100,000 km away from Earth on October 13, 2020.

For each ticket, randomly choose the speed the ship will travel, from 16 to 30 km/s. This will determine the duration for the trip to Mars and also the price. Make faster ships more expensive, ranging in price from $36 to $50 million. Double the price for round trips.

Experiment: tickets.go

Write a ticket generator in the Go Playground that makes use of variables, constants, `switch`, `if`, and `for`. It should also draw on the `fmt` and `math/rand` packages to display and align text and to generate random numbers.

When you're done, post your solution to the Get Programming with Go forums at forums.manning.com/forums/get-programming-with-go. If you get stuck, feel free to ask questions on the forums, or take a peek at Appendix A for my solution.

Capstone 2. The Vigenère cipher

The Vigenère cipher is a 16th century variant of Caesar cipher. For this challenge, you will write a program to decipher text using a keyword.

Before describing Vigenère cipher, allow me to reframe Caesar cipher, which you have already worked with. With Caesar cipher, a plain text message is ciphered by shifting each letter ahead by 3. The direction is reversed to decipher the resulting message.

Assign each English letter a numeric value, where A=0, B=1, all the way to Z=25. With this in mind, a shift by 3 can be represented by the letter D (D=3).

To decipher the text in Table 11.1, start with the letter 'L' and shift it by 'D'. Since L=11 and D=3, the result of 11-3 is 8 or the letter 'I'. Should you need to decipher the letter 'A', it should wrap around to become 'X', as you saw in the lesson 9.

Table 11.1 Caesar cipher

L	F	D	P	H	L	V	D	Z	L	F	R	Q	T	X	H	U	H	G
D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D

The Caesar cipher and ROT13 are susceptible to what's called *frequency analysis*. Letters that are occur frequently in the English language, such as 'E', will occur frequently in the ciphered text as well. By looking for patterns in the ciphered text the code can cracked.

To thwart would-be code crackers, the Vigenère cipher shifts each letter based on a repeating keyword, rather than a constant like 3 or 13. The keyword repeats until the end of the message, as shown for the keyword GOLANG in Table 11.2.

Now that you know what the Vigenère cipher is, you may notice that Vigenère with the keyword "D" is equivalent to the Caesar cipher. Likewise, ROT13 has a keyword of "N" (N=13). Of course longer keywords are needed to be of any benefit.

Table 11.2 Vigenère cipher

C	S	O	I	T	E	U	I	W	U	I	Z	N	S	R	O	C	N	K	F	D
G	O	L	A	N	G	G	O	L	A	N	G	G	O	L	A	N	G	G	O	L

Experiment: decipher.go

Write a program to decipher the ciphered text shown in Table 11.2. To keep it simple, all characters are upper case English letters for both the text and keyword.

```
cipherText := "CSOITEUIWUIZNSROCNKFD"
keyword := "GOLANG"
```

- The `strings.Repeat` [2: golang.org/pkg/strings/#Repeat] function may come in handy. Give it a try, but also complete this exercise without importing any packages other than `fmt` to print the deciphered message.
- Try this exercise using `range` in a loop and again without it. Remember that the `range` keyword splits a string into runes, whereas an index like `keyword[0]` results in a byte.

You can only perform operations on values of the same type, but you can convert one type to the other (string, byte, rune).

- To wrap around at the edges of the alphabet, the Caesar cipher exercise made use of a comparison. Solve this exercise without any `if` statements by using modulus (`%`).

If you recall, modulus gives the remainder of dividing two numbers. For example, `27 % 26` is `1`, keeping numbers within the 0-25 range. Be careful with negative numbers though, as `-3 % 26` is still `-3`.

After you complete the exercise, take a look at my solution at github.com/nathany/get-programming-with-go. How do they compare? Use the Go Playground's **[Share]** button and post a link to your solution in the Get Programming with Go forums forums.manning.com/forums/get-programming-with-go.

Ciphering text with Vigenère isn't any more difficult than deciphering text. Just add letters of the keyword to letters of the plain text message instead of subtracting.

Experiment: cipher.go

To send ciphered messages, write a program that ciphers plain text using a keyword.

```
plainText := "your message goes here"
keyword := "GOLANG"
```

Bonus: Rather than write your plain text message in upper case letters with no spaces, use the `strings.Replace` [5: golang.org/pkg/strings/#Replace] and `strings.ToUpper` [6: golang.org/pkg/strings/#ToUpper] functions to remove spaces and upper case the string before you cipher it.

Once you have ciphered a plain text message, check your work by deciphering the ciphered text with the same keyword.

Use the keyword `"GOLANG"` to cipher a message and post it to the forums for Get Programming with Go forums.manning.com/forums/get-programming-with-go.

Disclaimer: Vigenère cipher is all in good fun, but don't use it for important secrets. There are much more secure ways to send messages in the 21st century.

Capstone 3. Temperature tables

Write a program that displays temperature conversion tables. The tables should use equal signs (=) and pipes (|) to draw lines, with a header section.

```
======================
| ºC       | ºF       |
======================
| -40.0    | -40.0    |
| ...      | ...      |
======================
```

The program should draw two tables. The first table has two columns, with ºC in the first column and ºF in the second column. Loop from -40ºC through 100ºC in steps of 5º, and fill in both columns by using a temperature conversion function.

After completing one table, implement a second table with the columns reversed, converting from ºF to ºC.

Drawing lines and padding values is code that can be reused for any data that needs to be displayed in a 2-column table. Use functions to separate the table drawing code from the code that calculates the temperatures for each row.

Implement a `drawTable` function that takes a first-class function as a parameter and calls it to get data for each row drawn. Passing a different function to `drawTable` should result in different data being displayed.

Capstone 4. A slice of life

This section will step you through the process of building a simulation of underpopulation, overpopulation, and reproduction called Conway's Game of Life. While I will provide some guidance, you will need to write most of the code yourself. My own solution can be found with the source code at github.com/nathany/get-programming-with-go.

John Conway's Game of Life is a simulation played out on a two dimensional grid of cells. As such, this challenge focuses on slices.

In each generation, cells live or die based on their surrounding neighbors. Each cell has eight neighbors, which are adjacent in the horizontal, vertical, and diagonal directions.

- A live cell with less than two live neighbors dies.
- A live cell with two or three live neighbors lives on to the next generation.
- A live cell with more than three live neighbors dies.
- A dead cell with exactly three live neighbors becomes a live cell.

4.1. A new universe

For your first implementation of Game of Life, limit the universe to a fixed size. Decide on the dimensions of the grid and define a couple of constants.

```
const (
    width  = 80
    height = 15
)
```

Next, define a `Universe` type to hold a two-dimensional field of cells. With a Boolean type, each cell will be either dead (`false`) or alive (`true`).

```
type Universe [][]bool
```

Uses slices rather than arrays so that a universe can be shared with, and modified by, functions or methods.

Lesson 26 will introduce pointers, which provide an alternative, allowing you to directly share arrays with functions and methods.

Write a `NewUniverse` function that uses `make` to allocate and return a `Universe` with `Height` rows and `Width` columns per row.

```
func NewUniverse() Universe
```

Freshly allocated slices will default to the zero value which is `false`, so the universe begins empty.

4.2. Looking at the universe

Write a method to print a universe to the screen using the `fmt` package. Represent live cells with an asterisk and dead cells with a space. Be sure to move to a new line after printing each row.

```
func (u Universe) Show()
```

Write a `main` function to create a `NewUniverse` and `Show` it. Before continuing, be sure that you can run your program, even though the universe is empty.

4.3. Seeding live cells

Write a `Seed` method that randomly sets approximately 25% of the cells to alive (`true`).

```
func (u Universe) Seed()
```

Remember to import `math/rand` to use the `Intn` function. When you're done, update `main` to populate the universe with `Seed` and display your handiwork with `Show`.

4.4. Implementing the game rules

To implement the rules, break them down into three steps, each which can be a method:

- A way to determine if a cell is alive.
- The ability to count the number of live neighbors.
- The logic to determine if a cell should be alive or dead in the next generation.

4.4.1. Dead or alive?

It should be easy to determine whether a cell is dead or alive. Just lookup a cell in the universe slice. If the boolean is `true` then the cell is alive.

Write an `Alive` method on the `Universe` type with the following signature:

```
func (u Universe) Alive(x, y int) bool
```

A complication arises when the cell is outside of the universe. Is (-1,-1) dead or alive? On an 80x15 grid, is (80,15) dead or alive?

To address this, make the universe wrap around. The neighbor above (0,0) will be (0,14) instead of (0,-1), which can be calculated by adding `Height` to the row. In case row exceeds the `Height` of the grid, you can turn to the modulus operator (`%`) that was previously used for leap year calculations. Just use % to divide row by Height and keep the remainder. The same goes for `Width`.

4.4.2. Counting neighbors

Write a method to count the number of live neighbors for a given cell, from 0 to 8. Rather than access the universe data directly, use the `Alive` method so that the universe wraps around.

```
func (u Universe) Neighbors(x, y int) int
```

Be sure to only count adjacent neighbors and not the cell in question.

4.4.3. The game logic

Now that you can determine if a cell has two, three, or more neighbors, you can implement the rules. Write a `Next` method to do this.

```
func (u Universe) Next(x, y int) bool
```

Don't modify the universe directly, instead return whether the cell should be dead or alive in the next generation.

To to reiterate, the game rules are as follows:

- A live cell with less than two live neighbors dies.
- A live cell with two or three live neighbors lives on to the next generation.
- A live cell with more than three live neighbors dies.
- A dead cell with exactly three live neighbors becomes a live cell.

4.5. Parallel universe

To complete the simulation, you need to step through each cell in the universe and determine what its `Next` state should be.

There is one catch. When counting neighbors, it should be based on the previous state of the universe. If you modify the universe directly, those changes will influence the neighbor counts for surrounding cells.

A simple solution is to create two universes of the same size. Read through universe A while setting cells in universe B. Write a `Step` function to perform this operation.

```
func Step(a, b Universe)
```

Once universe B holds the next generation, you can swap universes and repeat.

```
a, b = b, a
```

To clear the screen before displaying a new generation, print `"\x0c"`, which is a special ANSI escape

sequence. Then display the universe and use the `Sleep` function from the `time` package to slow down the animation.

Now you should have everything you need to write a complete Game of Life and run it in The Go Playground.

Experiment: life.go

Build and run Conway's Game of Life.

Share a Playground link to your solution in the Manning forums. forums.manning.com/forums/get-programming-with-go

Capstone 5. Martian animal sanctuary

In the distant future, humankind may be able to comfortably live on what is currently a dusty red planet. Mars is further from the Sun, and therefore much colder. Warming up the planet could be the first step in *terraforming* the climate and surface of Mars. Once water begins to flow and plants begin to grow, organisms can be introduced.

> "Tropical trees can be planted; insects and some small animals can be introduced. Humans will still need gas masks to provide oxygen and prevent high levels of carbon dioxide in the lungs."
>
> — Leonard David, Mars: Our Future On The Red Planet

Right now the Martian atmosphere is approximately 96% carbon dioxide. [7: Atmosphere of Mars en.wikipedia.org/wiki/Atmosphere_of_Mars] It could take a very, very long time to change that. Mars will remain a different world.

Now it's time to use your imagination. What do you think would happen if an ark full of Earth animals were introduced to a terraformed Mars? What lifeforms might spring forth as the climate adjusts to support life?

Your task is to create a simulation of the first animal sanctuary on Mars. Make a few types of animals. Each animal should have a name and adhere to the `Stringer` interface to return their name.

Every animal should have methods to move and eat. The move method should return a description of the movement. The eat method should return the name of a random food that the animal likes.

Implement a day/night cycle and run the simulation for three 24-hour sols (72 hours). All the animals should sleep from sunset until sunrise. For every hour of the day, pick an animal at random to perform a random action (move or eat). For every action, print out a description of what the animal did.

Your implementation should make use of structures and interfaces.

Capstone 6. Sudoku rules

Sudoku is a logic puzzle that takes place on a 9x9 grid. Each square can contain a digit from 1 through 9. The number zero indicates a square that is empty.

The grid is divided into nine subregions that are 3x3 each. When placing a digit, it must adhere to certain constraints. The digit being placed may not already appear in:

- the horizontal row it is placed in
- the vertical column it is placed in
- the 3x3 subregion that it is placed in

Use a fixed-size (9x9) array to hold the Sudoku grid. If a function or method needs to modify the array, remember that you need to pass the array with a pointer.

Implement a method to set a digit at a specific location. This method should return an error if placing the digit does not adhere to the rules.

Also implement a method to clear a digit from a square. This method need not adhere to these constraints, as several squares may be empty (zero).

Sudoku puzzles begin with some digits already set. Write a constructor function to prepare the Sudoku puzzle, and use a composite literal to specify the initial values. Here is one possible example:

Listing 29.1 New Sudoku grid: sudoku.go

```
s := NewSudoku([rows][columns]int8{
    {5, 3, 0, 0, 7, 0, 0, 0, 0},
    {6, 0, 0, 1, 9, 5, 0, 0, 0},
    {0, 9, 8, 0, 0, 0, 0, 6, 0},
    {8, 0, 0, 0, 6, 0, 0, 0, 3},
    {4, 0, 0, 8, 0, 3, 0, 0, 1},
    {7, 0, 0, 0, 2, 0, 0, 0, 6},
    {0, 6, 0, 0, 0, 0, 2, 8, 0},
    {0, 0, 0, 4, 1, 9, 0, 0, 5},
    {0, 0, 0, 0, 8, 0, 0, 7, 9},
})
```

The starting digits are fixed in place and may not be overwritten or cleared. Modify your program so that it can identify which digits are fixed and which are penciled in. Add a validation that causes set and clear to return an error for any of the fixed digits. The digits that are initially zero may be set, overwritten, and cleared.

You don’t need to write a Sudoku solver for this exercise, but be sure to test that all the rules are implemented correctly.