Lesson 2.1 - Understanding Strings in Go, Formatted Output

This lesson explains how strings are represented in Go using UTF-8 encoding and how to work with them. It also covers using the **fmt** package for formatted output and the **strings** package for common string manipulations.

Key Takeaways

- Go strings are sequences of UTF-8 encoded runes, allowing them to represent characters from various languages.
- The built-in **strings** package offers functions for operations like repeating strings (strings.Repeat).
- Go strings are immutable, meaning their contents cannot be changed after creation. Operations that appear to modify a string actually create a new one.
- When iterating over a string using **for range**, you get the rune and its starting byte index, which is important for handling multi-byte characters correctly.

Key Concepts

- **UTF-8:** A variable-width character encoding capable of encoding all possible characters (code points) in Unicode. Go uses UTF-8 for its string representation.
- **Rune:** An alias for int32 in Go, representing a Unicode code point. It is used to handle individual Unicode characters correctly, especially when dealing with multi-byte characters.
- **String Immutability:** The property of strings where their value cannot be changed after they are created. Any operation that seems to modify a string results in the creation of a new string.
- **Formatted Output (fmt package):** Go's fmt package provides functions like Printf, Println, and Sprintf for producing formatted output to the console or creating formatted strings.

Lesson 2.2 - Calling REST APIs, HTTP Requests, JSON

This lesson teaches how to make HTTP requests to REST APIs in Go using the **net/http** package and how to work with JSON data using the **encoding/json** package. It covers making GET requests, understanding HTTP responses, and marshalling/unmarshalling JSON to and from Go structs.

Key Takeaways

- Go's **net/http** package provides functionality for making HTTP requests, including http.Get for making GET requests.
- HTTP responses contain a status code that indicates the result of the request (e.g., 200 for success).
- JSON is a common data format used in web APIs, and Go's **encoding/json** package is used to encode Go data structures into JSON and decode JSON into Go data structures.
- Go structs are often used to represent JSON data. For the encoding/json package to work with struct fields, they must be exported (start with a capital letter).

Key Concepts

- **REST API (Representational State Transfer Application Programming Interface):** An architectural style for building web services that uses standard HTTP methods for data retrieval and manipulation.
- **JSON (JavaScript Object Notation):** A lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate.
- **Struct Tag:** Metadata added to Go struct fields using backticks (`json:"fieldName"`) that provides instructions to the encoding/json package on how to map JSON keys to struct fields during marshalling and unmarshalling.
- **Marshalling/Unmarshalling:** The process of converting data between different formats. In Go, marshalling typically means converting Go data structures to JSON, and unmarshalling means converting JSON data to Go data structures.

Lesson 2.3 - Working with Files, Using Defer to Manage Resources, Error Handling

This lesson explains how to work with files in Go using the **os** package and emphasizes the importance of resource management using defer and proper error handling. It covers opening files, reading from them, and the idiomatic way to check and handle errors in Go.

Key Takeaways

- The **os** package provides functions like os.Open for opening files.
- The **defer** keyword is crucial for ensuring resources like open files are closed after the function completes, regardless of how it exits.
- Error handling in Go is done by checking the error value returned by functions. If the error is not nil, it indicates a problem that needs to be handled.
- File objects in Go provide methods for various operations, but all their fields are internal.

Key Concepts

- **File I/O (Input/Output):** Operations related to reading data from and writing data to files on a storage device.
- **defer Statement:** A statement in Go that schedules a function call to be executed after the surrounding function returns. It is commonly used for cleanup operations.
- **Error Handling:** The process of anticipating and responding to errors that may occur during the execution of a program. In Go, errors are often returned as values.

Lesson 2.4 - Composing `io.Reader` and `io.Writer`

This lesson explores the **io.Reader** and **io.Writer** interfaces in Go and how they can be composed to process data streams. It demonstrates reading from a compressed file using io.Reader composition and calculating a SHA1 signature using an io.Writer.

Key Takeaways

- **io.Reader** and **io.Writer** are fundamental interfaces in Go that define the Read and Write methods for handling streams of data.
- Go promotes composition of interfaces, allowing you to chain multiple readers and writers together to achieve complex data processing pipelines (e.g., decompressing and then hashing data).
- The **io.Copy** function is an efficient way to move data from an io.Reader to an io.Writer.
- The power of io.Reader and io.Writer lies in their abstraction, allowing them to work with various data sources and sinks (files, network connections, in-memory buffers) as long as they implement the respective interface.

Key Concepts

- **io.Reader Interface:** An interface that defines the Read(p []byte) (n int, err error) method, which reads up to len(p) bytes into the byte slice p.
- **io.Writer Interface:** An interface that defines the Write(p []byte) (n int, err error) method, which writes the bytes in p to the underlying data stream.
- **Interface Composition:** The design principle of building complex behavior by combining simpler interfaces. This allows for flexibility and reusability in Go's I/O operations.

Lesson 2.5 - Working with Slices, Slices Internals

This lesson provides a detailed explanation of slices in Go, including their creation, manipulation, and underlying structure. It covers the concepts of length, capacity, and the backing array, as well as different ways to iterate over slices and important considerations like value vs. pointer semantics in loops.

Key Takeaways

- Slices are dynamically sized views into an underlying array. They have a length (number of elements) and a capacity (size of the underlying array from the slice's starting point).
- The **append** function is used to add elements to a slice. If the capacity is insufficient, a new backing array with a larger capacity is allocated.
- Iterating over a slice with **for range** provides both the index and a copy of the value (value semantics). Modifying the copied value does not change the original slice element.
- To modify elements within a slice during iteration, you need to use the index to access the element directly or use a pointer receiver if the slice contains pointers.

Key Concepts

- **Slice:** A data structure that provides a dynamic view into a contiguous segment of an underlying array. Slices are more commonly used than arrays in Go.
- **Length:** The number of elements currently stored in a slice.
- **Capacity:** The maximum number of elements that can be stored in a slice's underlying array, starting from the slice's first element, without reallocation.
- **Backing Array:** The underlying array that a slice header points to. Multiple slices can share the same backing array, and changes in one slice might affect others if they share the same backing array and their ranges overlap.