Basic Syntax and Types in Go

This section will cover the basic syntax and various types in Go, providing a solid foundation for understanding how to declare variables, use types, and work with Go's data structures.

1. Variables, Types, and Type Inference

In Go, variables are explicitly declared and used by the compiler to e.g., check type-correctness of function calls.

Variable Declaration:

- The `var` keyword is used to declare one or multiple variables.
- The type of the variable is declared after the variable name.
- Example: `var age int`

Short Variable Declaration:

- Go also supports short variable declaration syntax using `:=`, which infers the type based on the initializer value.
- Example: `name := "John"`

Multiple Variable Declaration:

- You can declare multiple variables at once.
- Example: `var x, y int = 1, 2`

2. Constants and `iota`

Constants in Go are declared like variables but with the `const` keyword. Constants can be character, string, boolean, or numeric values.

Iota:

- `iota` is a special constant in Go that simplifies constant definitions that increment by 1.

- Each time `iota` is used, it automatically increments by 1.

- Example:

```
 const (
    Sunday = iota
    Monday
    Tuesday
    // continues automatically
)
```

3. Basic Data Types

Go supports several basic data types, which include:

- **Integers:** `int` , `int8` , `int16` , `int32` , `int64` , `uint` , `uint8` , etc.
- **Floats:** `float32` , `float64`
- **Strings:** `string`
- **Booleans:** `bool`

4. Composite Types

Go also supports composite types that group multiple values into a single value:

- **Arrays:**
  - Fixed length.
  - Example: `var a [5]int`
- **Slices:**
  - Dynamic length, more common than arrays.
  - Example: `s := []int{1, 2, 3}`
- **Maps:**
  - Key-value pairs.
  - Example: `m := map[string]int{"foo": 42}`
- **Structs:**
  - Collection of fields.
  - Example:

    ```
     type Person struct {
        Name string
        Age  int
    }
    p := Person{Name: "Alice", Age: 30}
    ```

These basic elements form the building blocks for constructing more complex programs and data structures in Go. Understanding them will help you in managing data effectively as you build Go applications.