Go Treat Yo-self. A Go desk reference by Sean Eyre (Twitter @oni_49) and Stephen Semmelroth (@diodepack).

==========

File types and packages.

Files containing go code end in `.go`, for example `hello_world.go`. At the top of every file there is a package name. Capital names are intended for export outside of the immediate project. Lower case names are "private" packages. The main package indicates that it is the main executable.

```
package main  // This .go file is the main executable; contains main()
package My_math  // This math package is intended for outside use
package private_objects  // This package is intended for internal use

import "name"  // Import package "name",
               // package names must match the end of their import path.
import "math/rand"  // Import just rand from math
import n "name"  // Import package name as "n" (use via n.method() )
import m "math"  // Import math as m
```

==========

Generic program structure

Go programs have the following generic structure, for more details on individual elements see further below.

```
package main  // Indicates this is the main executable

import "fmt"  // All imports are in quotations

func main() {  // Main method
   statement  // Placeholder for a real statement
   stmt  // Abbreviation for the above in this cheat sheet
}
```

Throughout most of this document's examples, *{ multiple_statements }* can also be substituted for *stmt*

==========

Statements

In Go, statements do not end with a semi-colon but with a new-line. You can use semi-colons to place multiple statements on one line, but this is usually only done for compound statements such as a statement embedded in a comparison.

```
var a int = 10  // A simple statement
fmt.Println("The value of a is", a)

if a := 10; a > 5 {  // Example multi-statement line.
   fmt.Println("a is greater than 5") // Must use double quotes
}

// Split line statements
s := `This is a
Multi-line string`  // Use backtick, not single-quote

a := b + c +  // Operator without a trailing operand automatically
searches the next line
   d + e + f
```

==========

Operators

Operator	Description
struct.member	Structure member reference
!; ^	Logical NOT, Bitwise NOT
++; --	Inc/decrement
**pointer; &name*	Indirection via pointer, address an object
type(expr)	Cast *expr* to *type*
*; /; %	Multiplication/ Division/ Modulo (checks for remainder after division)
+; -	Addition/ Subtraction
<<; >>	Bit shift left/right; each shift is x2 or /2
<; >	Less/greater than
<=; >=;	Less/greater than or equal
==; !=	Equal/ not equal to
&	Bitwise AND; 1 & 1 = 1, 1 & 0 = 0, 0 & 0 = 0
^	Bitwise XOR; 1 ^ 1 = 0, 1 ^ 0 = 1, 0 ^ 0 = 0 Also Bitwise NOT ^1100 -> 0011
|	Bitwise OR; 1 | 1 = 1, 1 | 0 = 1, 0 | 0 = 0
&^	Bit clear (AND NOT)
&&	Logical AND
||	Logical OR
=; :=; +=; -=; *=	Assignment; type implicit assignment; modify and assign
,	Evaluations separator
<-	Send / receive operator (for channels)
_ (underscore)	Drop returned value

Examples:

```
4 << 2  // Shift decimal 4 ("100" in binary) two bits left;
        // Returns "10000" or 16 in decimal; Single shift left is a x2
        //   Single shift right is a /2

var a, b, c int = 1, 2, 3  // Link together like statements with ","
```

==========

Data types

Go is a statically typed language. All variables and constants must have a type which does not change. A variable either has a declared type or an inferred type:

```
var name type  // Declare a variable of type
name = value  // Assign value into name

var x int
x = 25

var a, b, c int = 1, 2, 3  // Declare and assign multiple variables at
once

name := value  // When the := assignment operator is used,
               // Go creates a variable (not constant)
               // and sets its type inferred from the type of the value.
pi := 3.14  // Create a float variable named "pi" with value 3.14
a, b, c := 1, 2,3  // assign and infer type for multiple variables

const name type = value  // declare constant of type with name and value
const k int = 5
```

```
// Declare multiple variables (or constants) at once:
var (
   r int = 10
   pi float32 = 3.14
)
```

Declared variables must be used or it will produce a compile time error. Variables instantiated (declared) but not initialized (assigned) are implicitly assigned to the zero value of that data type (0 for ints, 0.0 for floats, an empty string, an array of zero values, etc)

```
var x int;
fmt.Println(x)  // Prints 0.
```

Data types and sizes:

Name	Size	Desc
bool	1 bit	True or False value
string	Multiples of 32 bits	*Immutable* sequence of *bytes (NOT characters);* doesn't have to be ASCII or UTF-8
int	Varies	Alias for your computer's default size of integer (can be int8, int16, int32, or int64)
Int32 / int64	32 bits, 64 bits	Signed integers of 32 or 64 bits
uint	Varies	Alias for your computer's default size of unsigned integer (uint8 – uint64)
uint32 / uint64	32 bits, 64 bits	Unsigned integer of 32 or 64 bits
byte	8 bits	Alias for int8
rune	32 bits	Symbols; Alias for an int32 storing a unicode code point
float32 / float64	32 bit, 64 bit	IEEE 754 floating-point number
complex64 / complex128	64 bits, 128 bits	Complex number made of floating-point numbers representing real and imaginary components

Type conversions:

Go does not conduct any implicit type conversions, so the type you expect to receive back must be explicitly stated

```
var i int = 42
f := float32(i)  // Case I to a floating-point number. Create f, infer it
                 // is a float, and set it to 42.0
u= uint(i)

var (
   r int = 10
   pi float32 = 3.14
)
area := float32(r*r) * pi  // Area will be a float
```

When you add integers together, they need to have the same bit length, cast to the same type first.

==========

Arrays

Arrays in Go are fixed length. Their contents can change and they can be sub-referenced.

```
var name [n]type  // Declare an array of type with length n labled name
var name [n]type = [n]type{value, value, …}  // Declare an array of type
                                    // with length n and initialize values

var counting_numbers [9]int
var counting_numbers = [9]int{1, 2, 3, 4, 5, 6, 7, 8, 9}

my_array := [3]int{1,2,3}  // Declare an array with inferred type
my_array := [...]int{1,2,3,4,5}  // Declare an array and infer length
(ellipsis tells compiler to infer length)

// When you sub-reference (slice!) an array, a slice is returned (fact
check)
my_array[3]  // Reference the third index of my_array
my_array[:3]  // Receive a slice back with indexes 0-2 of my_array (In
mathematical notation range (0,3] )
my_array[3:5]  // Slice with indexes 3 and 4 (range (3,5] )
my_array[3:]  // Slice with everything on and after index three
```

==========

Slices

Slices are arrays of dynamic length (Or rather, Arrays are slices of static length)

```
var name []type  // Declare a slice, leave the length of the array blank

var counting_numbers []int  //Instantiate a slice but do not initialize
var counting_numbers []int = []int{1,2,3}  //Instantiate and initialize
counting_numbers := []int{1,2,3}  // Inferred type declaration to
                                  // instantiate and initialize
```

Slices can be sub-referenced (sliced!) in the same way as arrays.

==========

Runes

Runes are 32-bit integers that store a "unicode code point." All Go source code is UTF-8 before it's compiled, but strings and runes are bytes or numeric pointers under the hood. Runes are initialized by assigning a UTF-8 string literal of length one (a single character) and are then stored as a 32-bit numerical representation of that UTF-8 character's index.

```
var name rune = char
var my_char rune = '¥'

my_char := '¥'  // Declare a rune via inferred type
char_2 := 'A'

char_array := [...]rune{'n','o','t','a','s','t','r','i','n','g'}
var char_slice []rune = []rune{'s','t','i','l','l','n','o', t'}
```

==========

Loops

In go, no one can hear you scream. There are only for-loops. There is no do-while, because there is no while. There is only for. For is all. For loops are made using an index (often with assignment), a condition, and an increment

```
for index := 0; index < 10; index++ {
   fmt.Println("I'm a classic for loop!")
}

//Alternatively, use a range
for index, element := range some_iterable {
   multiple_statements
}
```

You CAN break up the three components of a for loop, even placing them inside the loop itself.

```
index := 0
for index < 10{
   multiple_statements
   index++
}

//Or write it like a proper while loop, only it's not looping "while
true" so you might occasionally have to adjust your logic accordingly
index := true
for index {
   fmt.Println("I'm in danger!")
   index = false
}

idx := 0
for idx < 10{
   multiple_statement
   idx++
}

// Hurt your friends and instructors
idx := 0
for {  // Infinite loop
   multiple_statements
   idx++  // Not really required if there's no exit condition, is there?
}

stop := false
for {
   if stop {
      break
   }
   stop = !stop
}
```

==========

Range

Range provides a way to enumerate any interable variable and will generate its subcomponents in order.

Iterable	Range produces
Array	Index and element at index
Slice	Index and element at index
Map	Key and value (or just key)
String	Index and rune

Even though strings are immutable lists of bytes, when you execute range on a string, IT PRODUCES RUNES

```
counter := []int{4,5,6,7,8}  // A slice of numbers
for i, e := range counter {
   fmt.Println("Index", i, "has element", e)
}
// If you don't need i or don't need e
for _, e := range counter {
   multiple_statements
}
for i, _:= range counter {
   multiple_statements
}

dict := map[string]string{"Tired" : "The daily state of a programmer",
                          "Code": "The product of a programmer",
                          "Coffee" : "Transforms into code",
                         }

for k, v := range dict {
   fmt.Println("The key is ", k, " and the value is ", v)
}

for k := range dict {  // Iterate over just keys
   multiple_statements
}

str := "I'm a fuzzy unicorn!"
for I, r := range str {
   fmt.Println("Byte index: ", i, " rune (NOT BYTE): ", r)
}
```

Since maps are not ordered (see below) range over their keys produces inconsistent order.

==========

Maps

Maps are mappings from keys to values. Maps are unordered.

```
var my_map  map[key_type]value_type
```

The best way to initialize a map is using `make()`, but you can do it directly

```
my_map = make(map[key_type]value_type)  // This must match the
                                        // instantiation types
// Or immediately initialize
my_map = map[key_type]value_type{"Key_1": value_1, "K_2", v_2}
// Or instantiate and initialize
dict := map[string]string{"Tired" : "The daily state of a programmer",
                          "Code": "The product of a programmer",
                          "Coffee" : "Transforms into code",
                         }
```

```
// OR
var dict map[string]string = map[string]string {
                        "Tired" : "The daily state of a programmer",
                        "Code": "The product of a programmer",
                        "Coffee" : "Transforms into code",
                        }
// When broken into several lines each ending with a trailing comma
```

From there, you can assign and index values within the map using the keys

```
m:= make(map[string]int)
m["scouter's broken"] = 9001
fmt.Println(m["scouter's broken"])  // Will print 9001

delete(m, "key to delete")  // Delete a key from map m

// Reference a key directly to determine if it's in the map,
// catch the error thrown
value, okay := m["Non_existant_key"]  // Okay will be equivalent to true
                                      // if the key is present
```

Note: the `something, ok :=` pattern is referred to as the "comma okay" idiom in Go and is essentially used to "check if something is okay"

==========

Conditional Statements

If-statements *always* have blocks of code surrounded by { }, regardless of the simplicity of the statement. Variables initialized in the if-statement or its code block are limited to the scope of the if-statement or code block.

```
i := 5
if i < 5 {
   multiple_statements
} else if i == 1 {
   multiple_statements
} else {
   multiple_statements
}

x, y := 5, 6
if i = x*y % 2; i == 0 {  // You may include ONE assignment statement in
                          // the if-statement
   fmt.Println("i is even!", i) // i declared in the if-stmt so works
} else {
   fmt.Println("i is odd!", i)  // i declared in the if-stmt so works
}

a := 5
if a == 5 {
       i := 10
       fmt.Println(I * a)
} else {
       fmt.Println(I * a)  // i is not accessible here because it is
                           // part of the above CODE BLOCK's scope,
                           // not the scope of the if statement.
}
```

```
// Use an embedded statement to check type via assertion
var check interface{}  // Needed to make type assertion
check = "String?"
if _, ok = check.(string); ok {
       // Only reach here if there's not been a panic created by the
       // .(string) type assertion
       fmt.Println(strings.Replace(str, "?", "!"))  // Prints "String!"
}
```

Switch

Switches may either operate on discrete cases or conditional cases.

```
// Discrete switch
letter := "a"
switch a {  // You may include ONE assignment statement here
   case "a":
      statement
      // THERE IS NO FALL THROUGH IN GO, breaks are implicit
   case "b":
      statement
      break  // You can still make use of an explicit break
   default:
       statement
}

// Conditional switch
number := 5
swtich {
   case number == 5:
      fmt.Println("FIVE!")
      fallthrough  // If you want fall through for some reason
   case number > 5:
      fmt.Printn("Still got here!")
   case number < 5:
      fmt.Println("Didn't get here -_-")
   default:
      statement
}
```

==========

Strings:

In Go, strings are immutable collections of bytes. Each index references a byte-value that is typically *rendered* as a UTF-8 compliant string during I/O but strings have *no obligation* to be UTF-8 compliant. It is a slice, but is *read-only.* Because of this, we can assign strings as collections of UTF-8 characters or as collections of bytes via hexadecimal.

```
var my_string string = "This is a string."
str_2 := "Another string"
str_3 := "\x6e\x6f\x74\x20\x75\x74\x66\x2d\x38"

//printing will attempt to render all as UTF-8
fmt.Println(str_3)

// View it as bytes with
fmt.Printf("%x", str_2)  // see below for printf formatting
```

Because they are slices of bytes, strings can be sub-referenced the same way.

```
str_3[0]  // First byte value of str_3
str_2[1:5]  // Bytes at 1-4 (inclusive, (1,5] )
str_2[8:]  // All bytes from 8 to the end
```

PrintF formatting

%d	integer
%f	float
%v	value
%s	string
%b	binary value
%c	Unicode
%e	scientific notation
%x	hexadecimal
%q	print UTF-8 characters and escape as hex all others
%T	type
%o	octal
%O	octal with O prefix
%U	Unicode format (U+)
%p	pointer address in hex (points to start of the variable)

==========

Functions

All Go functions are called by value except mutators which are explained below. Go passes a copy of the variable/value to the function. Declare functions OUTSIDE of main, except lambda functions

```
func name(var_1 type, var_2 type) return_type {
   Multiple_statements
   Return value  // Must match type of return_type
}

// Return multiple values
func name(var_1 int, var_2 int) (return_type, return_type_2) {
   multiple_statements
}
```

==========

Closures

Go allows functions to be used to enclose variables in functions in order to create functions such as generators. Every time a closure is called, it instantiates a variable and a fresh anonymous function to return.

```
func generate_int() func() int {  // Return a func that returns an int
   i := 0
   return func() int { i++; Return i }
}  // This creates a closure that generates the next int each call

next_int := generate_int()
fmt.Println(next_int())  // Prints "1"
fmt.Println(next_int())  // Prints "2"
```

==========

Pointers

Pointers store values of addresses. Unlike in C, pointer math does not exist in go. Functions receiving a pointer as a parameter do not modify the original pointer because functions are call by value (passes a copy of the value) except for mutators. Zero values for pointers are nil.

```
var p *type = new(type)  // declare a new pointer, p to a type

a := 10
p := &a  // Pointer to an int
```

==========

Structs

There are no objects in Go, but there are structs, which have values and can have methods. (There are "no objects", but structs meet most of the definition of an object.)

```
// Declare a new struct:
type name struct{  // New type is a struct of name
   value_1 type  // It holds value_1 of type
   value_2 type  // Can have multiple values of different types
   ...
   value_n type
}

coord := struct{ x, y int }{ 5, 4}  // Declare an "anonymous" structure
fmt.Println(coord.x) // Reference Values within the struct
```

Struct methods.

You can declare methods that operate on a structure, the structure is STILL copied on function call.

```
func (c coord) smoosh() int {
   return c.x + c.y
}
fmt.Println(c.smoosh())  // Makes a copy of c, adds both values, return
                         // the integer
```

Mutators

The only way you can call by reference is a mutator. This is a struct method that receives a pointer to a struct instead of just a structure and is the only time a function is called WITHOUT making a copy of the parameter.

```
func (c *coord) slope(b int) {
   c.y = c.x + b
}  // Morphs c such that y = mx + b
```

==========

Routines

Go routines are threads managed by Go, they are not managed by the OS. They are called using the keyword go

```
go my_func()  // Run my_func() as a routine
```

==========

Channels

Go allows sending messages between channels. Channels that are not initialized are nil. While nil, channels that are sent to or received from are blocked permanently. Channels that are blocked cannot be sent to or received from. Think of this as simple inter-process communications for routines

```
c := make(chan int)  // Make a channel that sends integers
c <- 9001  // Send an int into the channel

scouter := <- c // Read from c and store as new variable scouter

// Close a channel
close(c)
```

==========

References:

- https://golang.org/
- https://github.com/a8m/golang-cheat-sheet#arrays-slices-ranges
- https://itnext.io/googles-go-essentials-for-node-js-javascript-developers-6d71f08d2531
- https://github.com/golang/go/wiki/Switch
- https://blog.golang.org/strings
- https://gobyexample.com/
- https://medium.com/golangspec/import-declarations-in-go-8de0fd3ae8ff