

Go/CSP Cheat Sheet

Compiling and Running Go Programs

Building	<code>go build file.go</code> Creates an executable named file
Running	<code>go run file.go</code> Runs file.go as a script

Data Types

General Variables can be declared implicitly or explicitly
`var myInt int; //explicit`
`myInt := 1; //implicit`

`bool` Boolean type

`int` Integer type

`uint` Unsigned Integer type

`int32` 32-bit integer (equivalent to a rune)

`float32` 32 and 64 bit floating points
`float64`

`complex64` 64 and 128 bit complex numbers
`complex128`

`string` 64 bit floating points

`const` Constant type, similar to `#define` in C
ex. `const Pi = 3.14159`

Essential Libraries

`fmt` Contains format print and I/O functions (similar to C I/O functions)

`os` Contains system calls and program argument vector

`strings` Contains string functions

`sync` Contains synchronization functions

`flag` Contains tools for parsing command line flags

`ioutil` Contains utilities for file I/O

OS interface

General `os` functions can be imported with `import "os"` and accessed by `os.<FunctionName>`

`os.Args[n]` Access element *n* of the argument vector

`len(os.Args)` Number of arguments in argument vector

`os.Exit(n)` Exit with status *n*

`os.Environ()` Get array of strings containing environment

Strings, Slices, and Arrays

Array Fixed size contiguous memory
ex. `var buffer [10]int`

Slices Piece of an array (sharing memory with the array)
ex. `var myslice = buf[4:7]`

Strings Read-only slice of bytes (can be many formats, Unicode, UTF-8, ASCII, etc...)
ex. `var Str string = "str"`

String functions

General String functions can be imported with `import "strings"` and accessed by `strings.<FunctionName>`

`Contains` `Contains("123", "12")`
-Output is true.

`Split` `Split("1 2 3", " ")`
-Output is [1, 2, 3] in an array.

`Join` `Join(array, char)`
-Joins an array of strings (array), putting char between each string (char can be "")

`len` `len("Hello")`
-Outputs 5

`"four"[n]` Access index *n* of "four"

Channels

Spawning goroutines `go <functionname>`

Channels Channels are pipes that connect concurrent goroutines. Channels can be used to send values between goroutines.

Unbuffered Channels goroutine will block unless another goroutine is waiting to read from the channel

Buffered Channels Channel contains a buffer that can be written to without a goroutine waiting to read from the channel

I/O with channels
`channel <- 1`
-Write to channel
`myInt <- channel`
-Read from channel

```
select {  
case  
<I/O>:  
default:  
}
```

 Allows goroutine to wait on multiple channels without blocking (either performing writes or reads)

Wait Groups

General Allows for a parent goroutine to wait for a collection of other goroutines to finish

Declaring wait groups `var wg = &sync.WaitGroup{}`
`wait`
`groups`

Adding `wg.Add(n)`
-Add *n* threads to the wait group

Signaling `wg.Done()`
-Goroutine signals it is finished

Waiting `wg.Wait()`
-Allows main thread to wait until child routines are finished