go cheatsheet v0.1 (@bm4cs)

Variables

There's no such thing as a undefined variable in Go. Each data type has a well defined zero value.

```
var a int //set to 0
var b int = 10
var d, e, f bool
var (
    g int
    h string
    i int = 1234
    j, k, l bool
)
```

Inside a func can use *short variable declaration*:

```
m := 1
n, o := 2, 3
```

The blank identifer (for ignoring outputs):

```
a, _ := f()
```

Conditions

if

Go conditions uniquely supports an *initialisation statement*:

```
if err := f(x); err != nil {
    return err
}
```

Ternary operators are not supported, due to poor readability.

switch

- Do not fallthrough
- Unlike many langs supports switch-expressionless form

```
switch {
    case velocity == 0 || velocity == 1:
        //...
}
```

Loops

for

- The `for` keyword covers off all loop types in Go. No support for `while`, `do`, `until` loops exist.
- For loop `for i=0; i<10; i++`
- While loop `for i<10`
- Infinite loop `for`
- Enumerable iteration `for i, v := range s`

Strings

- Uses UTF-8 to encode unicode. Recall UTF-8 is a variable length encoding (i.e. it can use 1-byte or upto 4-bytes).
- Go allows you to treat strings at both the byte and character level.
- Are immutable (mutability is possible using a slice of bytes)
- Treated as sequences of bytes with arbitrary values, even a null byte, differentiating them from C strings
- At the character level, the unit of a character is called a `rune`. A `rune` is just an alias for a `int32`.
- Runes are presented by `range` when iterating over a string:
- Substrings `s[1:4]` (the second index hits the first byte after the substring)
- Substring shortcuts `s[:4]` and `s[4:]`
- Useful stdlib packages `strings`, `strconv` and `unicode`

Constants

- A unicode rune, `'\377' '\u266B' '\U0001F600'`
- Integers, as decimal `1337`, octal `01337`, or hex `0x1CAFE42`
- Floats `123.456e7`, `123e20`, `1.234`, `.71828`, `2.e7`, `.5e-10`
- Strings, any sequence of unicode runes or escape sequences, `"A string \U0001F600"`
- Booleans `true` or `false`
- The `iota` keyword is used to generate enumerations (starting at 0 and incrementing by 1)

Pointers

- Supports C like `*type` pointer type, `&` address of, and `*p` pointer indirection
- Pointers have a default value of `nil`
- Its illegal to point to an arbitrary (literal) address, or a constant
- When `nil`, pointer indirection causes a panic
- Can compare pointer addresses `p1 == p2`, or the values they point to `*p1 == *p2`
- The built-in `new()` allocates variable and returns a pointer to it `p := new(int)`

Functions

- Functions are first class objects in Go (i.e. can be assigned to a variable)
- Subsequent parameters of the same type can be grouped `func f(n, m int, s string)`
- Named parameters are NOT supported (i.e. must pass in the order specified)

- Variadic functions are supported `func f(s ...string)`
- Functions can return multiple values (types must be enclosed in pathentheses)
- Return values can be named, which get declared as variables within the scope of the function
- `defer` will queue a function call until the point where the calling function itself exits

Functions as values

```
func f1(s string) bool {
    return len(s) > 0
}

func f2(s string) bool {
    return len(s) < 4
}

var funcVar func(string) bool

func main() {
    funcVar = f1
    fmt.Println(funcVar("abcd"))
    funcVar = f2
    fmt.Println(funcVar("abcd"))
}
```

Anonymous functions

```
funcVar = func(s string) bool {
    return len(s) > 4
}
```

Its possible to evaluate the function literal after defining it, which can be useful when creating `goroutines` in a loop:

```
var result string = func() string {
    return "abcd"
}()
```

Function passing

```
func f1(s string) bool {
    return len(s) > 0
}

func f2(s string) bool {
    return len(s) < 4
}

func funcAsParam(s string, f func(string) bool) bool {
    return f(s + "abcd")
}

func main() {
    fmt.Println(funcAsParam("abcd", f1))
}
```

Closures

When a function literal is defined within another function.

Closures get access to the local variables (i.e. the call stack) of the outer function, even after the lifetime of the outer function.

```
func newClosure() func() {
    var a int
    return func() {
        fmt.Println(a)
        a++
    }
}

c := newClosure()
c()
c()
c()
```