

Basic Syntax

```
package main
import "fmt"
func main() {
    fmt.Println("Hello Go")
}
```

Packages

- package declaration at top of every source file
- executables are in package main
- convention: package name == last name of import path (import path math/rand => package rand)
- upper case identifier: exported (visible from other packages)
- Lower case identifier: private (not visible from other packages)

Operators

Arithmetic

- + addition
- subtraction
- * multiplication
- / quotient
- % remainder
- & bitwise and
- | bitwise or
- ^ bitwise xor
- &^ bit clear (and not)
- << left shift
- >> right shift

Comparison

- == equal
- != not equal
- < less than
- <= less than or equal
- > greater than
- >= greater than or equal

Operators (cont)

- > Logical
- && logical and
- || logical or
- ! logical not
- Other
- & address of / create pointer
- * dereference pointer
- <- send / receive operator

Functions

```
// a simple function
func functionName() {}
// function with parameters
func functionName (param1 string, param2 int) {}
// multiple parameters of the same type
func functionName (param1, param2 int) {}
// return type declaration
func functionName() int {
    return 42
}
// return multiple
func returnMulti() (int, string) {
    return 42, "foo bar"
}
var x, str = returnMulti()
// Return multiple named results simply by return
func returnMulti2() (n int, s string) {
    n = 42
    s = "foo bar"
    // n and s will be returned
    return
}
var x, str = returnMulti2()
Functions As Values And Closures
func main() {
    // assign a function to a name
```

Functions (cont)

```
> add := func(a, b int) int {
    return a + b
}
// use the name to call the function
fmt.Println(add(3, 4))
}
// Closures, lexically scoped: Functions can access values that were
// in scope when defining the function
func scope() func() int{
    outerVar := 2
    foo := func() int { return outerVar }
    return foo
}
func anotherScope() func() int{
    // won't compile because outerVar and foo not defined in this scope
    outerVar = 444
    return foo
}
// Closures: don't mutate outer vars, instead redefine them!
func outer() (func() int, int) {
    outerVar := 2
    inner := func() int {
        outerVar += 99 // attempt to mutate outerVar from outer scope
        return outerVar // => 101 (but outerVar is a newly redefined
        // variable visible only inside inner)
    }
    return inner, outerVar // => 101, 2 (outerVar is still 2, not mutated by foo!)
}
```

Variadic Functions



Functions (cont)

```
> func main() {
    fmt.Println(adder(1, 2, 3)) // 6
    fmt.Println(adder(9, 9)) // 18
    nums := []int{10, 20, 30}
    fmt.Println(adder(nums...)) // 60
}

// By using ... before the type name of the
// last parameter you can indicate that it takes
// zero or more of those parameters.
// The function is invoked like any other
// function except we can pass as many
// arguments as we want.
func adder(args ...int) int {
    total := 0
    for _, v := range args { // Iterates over the
        // arguments whatever the number.
        total += v
    }
    return total
}
```

Declarations

```
var foo int // declaration
// without initial.

var foo int = 42 // declaration
// with initial

var foo, bar int = 42, 1302 //
// declare and init

var foo = 42 // type omitted,
// will be inferred

foo := 42 // shorthand

const constant = "This is a
// constant"
```

Type Conversions

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
// alternative syntax
i := 42
f := float64(i)
u := uint(f)
```

Arrays, Slices, Ranges

Arrays

```
var a [10]int
// declare an int array with
// length 10. Array length is part
// of the type!
a[3] = 42 // set elements
i := a[3] // read elements
// declare and initialize
var a = [2]int{1, 2}
a := [2]int{1, 2} // shorthand
a := [...]int{1, 2} // elipsis
// -> Compiler figures out array
// length
```

Slices

```
var a []int // declare a slice -
// similar to an array, but length
// is unspecified
var a = []int{1, 2, 3, 4} //
// declare and initialize a slice
// (backed by the array given
// implicitly)
a := []int{1, 2, 3, 4} //
// shorthand
chars := []string{0:"a ",
1:"c ", 1:" b"} // ["a", " b",
// " c"]
var b = a[0:hi] // creates a
// slice (view of the array) from
// index lo to hi-1
var b = a[1:4] // slice from
// index 1 to 3
var b = a[:3] // missing low
// index implies 0
var b = a[3:] // missing high
// index implies len(a)
// create a slice with make
a = make([]byte, 5, 5) // first
// arg length, second capacity
a = make([]byte, 5) // capacity
// is optional
// create a slice from an array
x := [3]string{"Яка", " -
// Белка", " Стрелка"}
}
```

Arrays, Slices, Ranges (cont)

```
> s := x[:] // a slice referencing the storage
// of x
```

Built-in Types

```
bool
string
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64
uintptr
byte // alias for uint8
rune // alias for int32 ~ = a
// character (Unicode code point) -
// very Viking
float32 float64
complex64 complex128
```

Control structures

If

```
func main() {
    // Basic one
    if x > 0 {
        return x
    } else {
        return -x
    }
    // You can put one
    // statement before the condition
    if a := b + c; a < 42 {
        return a
    } else {
        return a - 42
    }
    // Type assertion inside
    if
        var val interface{}
        val = "foo"
        if str, ok := val.(string); ok {
            fmt.Println(str)
        }
}
```

Loops



Control structures (cont)

```
> // There only for, no while, no until
for i := 1; i < 10; i++ {
}
for ; i < 10; { // while - loop
}
for i < 10 { // omit semicolons
}
for { //omit the condition ~ while (true)
}
```

Switch

```
switch operatingSystem {
case "darwin":
    fmt.Println("Mac OS Hipster")
    // cases break automatically
case "linux":
    fmt.Println("Linux Geek")
default:
    // Windows, BSD, ...
    fmt.Println("Other")
}

// as with for and if, you can have an
assignment statement before the switch
value

switch os := runtime.GOOS; os {
case "darwin": ...
}
```



By [deleted]
cheatography.com/deleted-23330/

Published 7th September, 2015.
Last updated 12th May, 2016.
Page 3 of 3.

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
<http://crosswordcheats.com>