Predeclared Types and Declarations

I The Zero Value

* Go assigns a default zero value to any variable that is declared but not assigned a value
* Having an explicit zero value makes the code clearer and removes a source of errors found in C/C++

II Literals

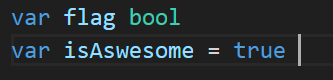
* It’s an explicitly specified number/character/string

1. Integers

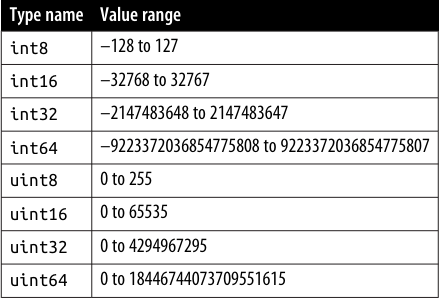
* Sequence of numbers : Integer literals are bas 10 by default but different prefixes are used to indicate other bases
* ***0b*** : binary (base 2)
* ***0o*** : octal (base 8)
* ***0x*** : hexadecimal (base 16)
* Go allows to put underscores in the middle of our literal : allows to group by thousands in base 10 : ***1\_234***
* Underscore has no effect on the value of the number (they can’t be at the beginning/end)
* We have to use them to improve readability
* ***Floating-point*** : has a decimal point to indicate the fractional portion of the value (can be positive and negative)
* Can gave an exponent specified with the letter ***e*** : ***6.03e23***
* We can write them in hexadecimal by using ***0x*** et ***p*** to indicate any exponent : 0x12.34p5
* ***Rune*** : represents a single character (is surrounded by single quotes)
* Single Unicode character : (‘a’)
* 8-bit octal numbers : (‘\141’)
* 8-bits hexadecimal numbers : (‘\x61’)
* 16-bit hexadecimal numbers : (‘\u0061’)
* 32-bit Unicode numbers : (‘\U00000061’)
* (‘\t’) : tabulation
* (‘\n’) : to go to the next line/skip one line
* (‘\\’) : to print \

1. Booleans

* Only two values : True/False



1. Numeric types
2. Integer types



* Special integer types
* Byte : uint8
* Int : int32(on 32-bit CPU)/int64(on 64-bit CPU)
* uint : same rules as int but is unsigned (always => 0)
* Choosing which integer to use
* Binary file : use the corresponding integer type
* Library function that should work with any integer type : use generic type parameter to represent any integer type
* In all other cases : Int
* Integer operators : +, -, \*, /, %
* The result of an integer division is an integer :

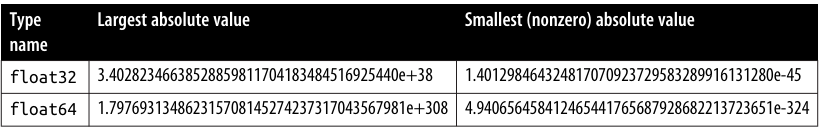
→ To get a floating-point result: we need to use a type conversion

* We can combine any operators with = to modify a variable
* Bit shift left ***(<<)***/right***(>>) : (0010 << 1 = 0100 ou 0010 >> 2 = 1000)***
* Bit masks : ***&*** (and), ***|*** (or)

→ ***^(XOR)***: binary operation that takes 2 bit patterns of equal length and perform the logical exclusive OR operation on each pair of corresponding bits

→ &^ (AND NOT)

1. Floating-point types



* Unless we have to be compatible with an existing format : use ***float64***
* We don’t have to worry about the difference in memory size unless we have used the profiler to determine that it’s a significant source of problems
* The bigger question is whether we should use floating-point numbers at all
* In many cases the answer is no : they can’t store every value but the nearest approximations
* Because they’re not exact they can only be used in situations where inexact values are acceptable

→ Graphics, statistics and specific operations

* We can use any standard mathematical/comparison operators ***EXCEPT %***
* Floating-point division has some interesting properties
* Dividing a non 0 floating point by 0 returns ***+Inf*** or ***-Inf*** (positive/negative infinity) depending on the sign of the number
* We don’t wanna use ***==*** and ***!=*** to compare floats
* Because of the inexact value of a float : 2 floats might not be equal when we think they should be
* Instead : define a maximum allowed variance and see if the difference between 2 floats is less than that

→ This value (epsilon) depends on our accuracy needs

1. Complex types

* ***Complex64*** : float32
* ***Compex128*** : float64



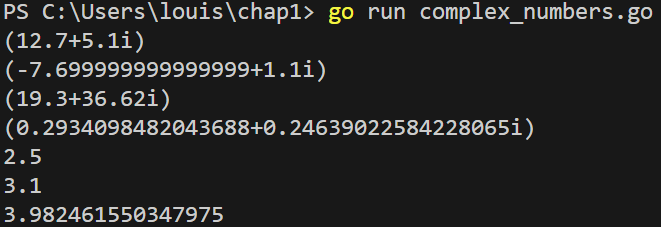
* Go got a few rules to determine the type of the value returned by complex
* Untyped constants/literals for both function parameters : untyped literal (complex128)
* Both values are float32 : complex64
* One value is float32 the other one is untyped that can fit with float32 : complex64
* Any other cases : complex128
* All the standard floating-point arithmetic operators work on complex numbers
* We can use == or != to compare them but they have the same precision limitations

→ It’s best to use the epsilon technique

* We can extract real/imaginary portions of a complex number with a real and imag built in functions
* The math/cmplx package has additional functions for manipulating complex128 values
* O value for both types of complex numbers has 0 assigned to both real/imaginary portions of the number



* Running this code gives us the following result :



* The imaginary portion of a complex number has ***i*** for suffix

1. Strings

* Zero value = ““
* We can put any Unicode character into a string
* We can compare them : ==, !=
* We can order them : <, >, <=, >=
* They are concatenated using the + operator
* Are immutable : we can reassign the value of a string variable but we can’t change the value of the string that is assigned to it
* If we are referring to a character we have to use the rune type not the int32 type