Data Types and Composite Types

Objectives

- Gain an introduction to type system
- Understand:
  - Integers
  - Floats
  - Strings
  - Key type manipulation packages
  - Arrays
  - Slices
  - Maps

Go Types

- Go’s types fall into four categories:
  - basic types
    - numbers, strings, and booleans
  - aggregate types
    - arrays and structs
  - reference types
    - pointers, slices, maps, functions, and channels
  - interface types
    - a way to specify the behavior of an object

Integers

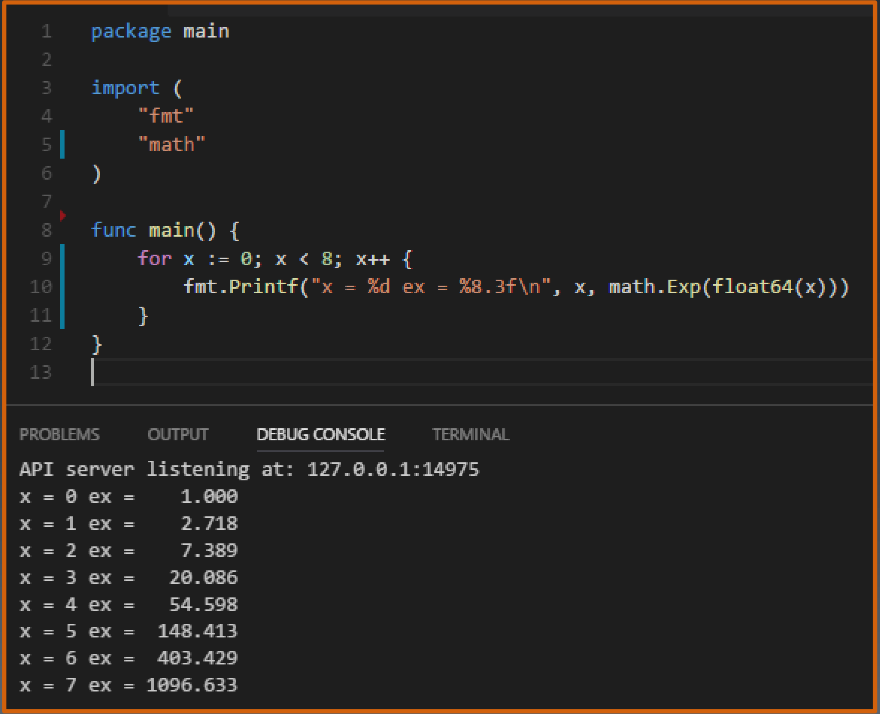
Printed with fmt functions using:
%d %o %x and %b
decimal, octal, hex and binary respectively

- Go supports several sizes of signed and unsigned integers
- There are four distinct sizes of signed/unsigned integers
  - int8/uint8 - 8 bits
  - int16/uint16 - 16 bits
  - int32/uint32 - 32 bits
  - int64/uint64 - 64 bits
  - Unsigneds are avoided and used only when bitwise operations are required on binary data
- There are also (most efficient size for signed and unsigned integers on a particular platform)
  - int
  - uint
  - Both have the same size (either 32 or 64 bits)
  - Different compilers may make different choices even on identical hardware
- rune is a synonym for int32
  - Indicates that a value is a Unicode code point
  - The two names may be used interchangeably
- byte is a synonym for uint8
  - emphasizes that the value is a piece of raw data rather than a small numeric quantity
- uintptr has unspecified width but is sufficient to hold all the bits of a pointer value
  - used only for low-level programming (at the boundary of a Go program and a C library or OS)
- int, uint, and uintptr are different types from their explicitly sized siblings

Floating Point

- Go provides two sizes of floating-point numbers
  - float32
  - float64
- IEEE 754 standard implemented by all modern CPUs
- The math package provides float limits
  - math.MaxFloat32 (about 3.4e38)
  - math.MaxFloat64 (about 1.8e308)
  - math.MinFloat32 (about 1.4e-45)
  - math.MinFloat64 (about 4.9e-324)
- Precision
  - float32 provides approximately 6 decimal digits of precision
  - float64 provides approximately 15 decimal digits of precision
- float64 should be preferred to avoid cumulative errors
- Scientific notation is supported
  - const Avogadro = 6.02214129e23
  - const Planck = 6.62606957e-34
- math.NaN() equates to the result operations such as 0/0 or math.Sqrt(-1)
  - %f will display "NaN"

Printed with fmt functions using:
%g, %e, %f
General, exponent and normal respectively

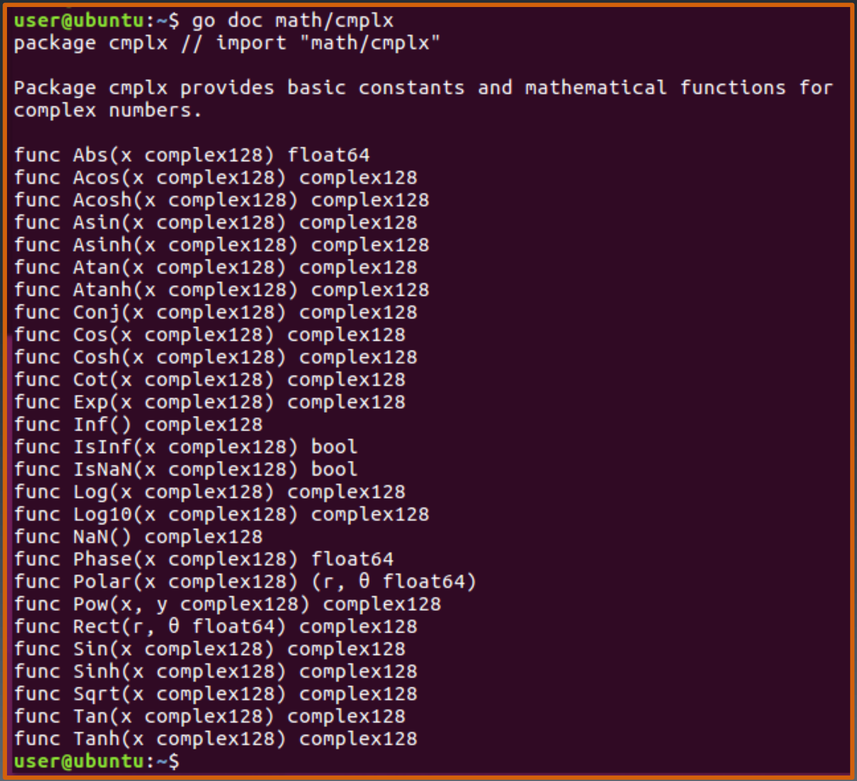


```go
package main

import (
    "fmt"
    "math"
)

func main() {
    for x := 0; x < 8; x++ {
        fmt.Printf("x = %d ex = %8.3f\n", x, math.Exp(float64(x)))
    }
}

```

```
API server listening at: 127.0.0.1:14975
x = 0 ex =    1.000
x = 1 ex =    2.718
x = 2 ex =    7.389
x = 3 ex =   20.086
x = 4 ex =   54.598
x = 5 ex =  148.413
x = 6 ex =  403.429
x = 7 ex = 1096.633
```

Complex Numbers

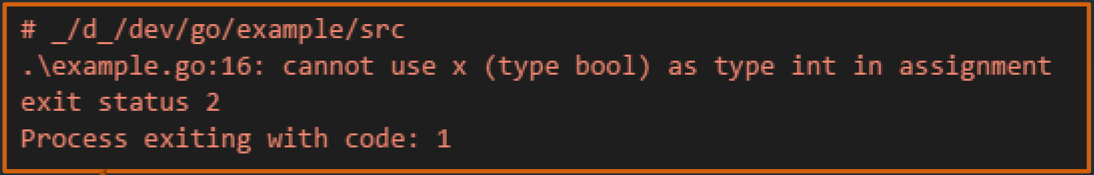
- Go provides two sizes of complex numbers
  - complex64
  - complex128
- These have real and imaginary components which are float32 and float64 respectively
- The built-in function complex creates a complex number from its real and imaginary components
  - var x complex128 = complex(1, 2)
  - Also using the "i" suffix
    - x := 1 + 2i
    - y := 3 + 4i
- The built-in real and imag functions extract those components
  - fmt.Println(real(x*y))
  - fmt.Println(imag(x*y))
- The standard math.cmplx package contains math functions for complex numbers



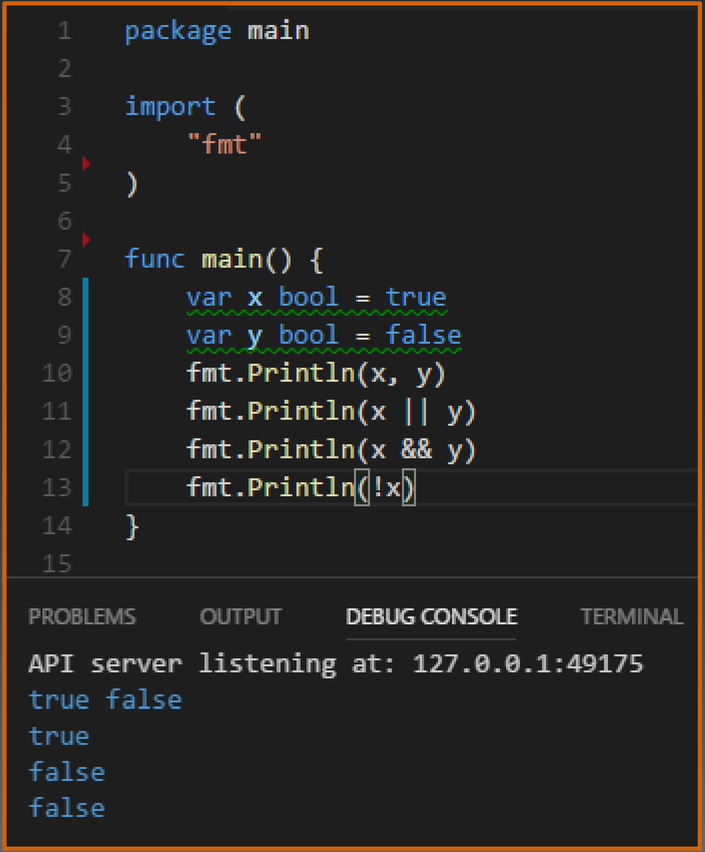
```
user@ubuntu:~$ go doc math/cmplx
package cmplx // import "math/cmplx"

Package cmplx provides basic constants and mathematical functions for
complex numbers.

func Abs(x complex128) float64
func Acos(x complex128) complex128
func Acosh(x complex128) complex128
func Asin(x complex128) complex128
func Asinh(x complex128) complex128
func Atan(x complex128) complex128
func Atanh(x complex128) complex128
func Conj(x complex128) complex128
func Cos(x complex128) complex128
func Cosh(x complex128) complex128
func Cot(x complex128) complex128
func Exp(x complex128) complex128
func Inf() complex128
func IsInf(x complex128) bool
func IsNaN(x complex128) bool
func Log(x complex128) complex128
func Log10(x complex128) complex128
func NaN() complex128
func Phase(x complex128) float64
func Polar(x complex128) (r, θ float64)
func Pow(x, y complex128) complex128
func Rect(r, θ float64) complex128
func Sin(x complex128) complex128
func Sinh(x complex128) complex128
func Sqrt(x complex128) complex128
func Tan(x complex128) complex128
func Tanh(x complex128) complex128
user@ubuntu:~$
```

Booleans

- Booleans are declared:
  - bool
- Two possible values:
  - true
  - false
- There is no implicit conversion of a bool to a numeric
- Conditions in if and for statements are Booleans
- Comparison operators like == and < produce Booleans
- The unary operator ! is logical negation
  - "x == true" can be simplified to "x" ("x" being preferred)
- Boolean values can be combined && (AND) and | | (OR)
  - && has precedence
  - short-circuit behavior



```
# _/d_/dev/go/example/src
.\example.go:16: cannot use x (type bool) as type int in assignment
exit status 2
Process exiting with code: 1
```

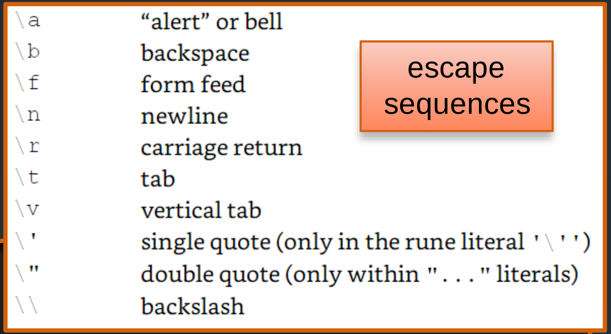


```go
package main

import (
	"fmt"
)

func main() {
	var x bool = true
	var y bool = false
	fmt.Println(x, y)
	fmt.Println(x || y)
	fmt.Println(x && y)
	fmt.Println(!x)
}
```

```
API server listening at: 127.0.0.1:49175
true false
true
false
false
```

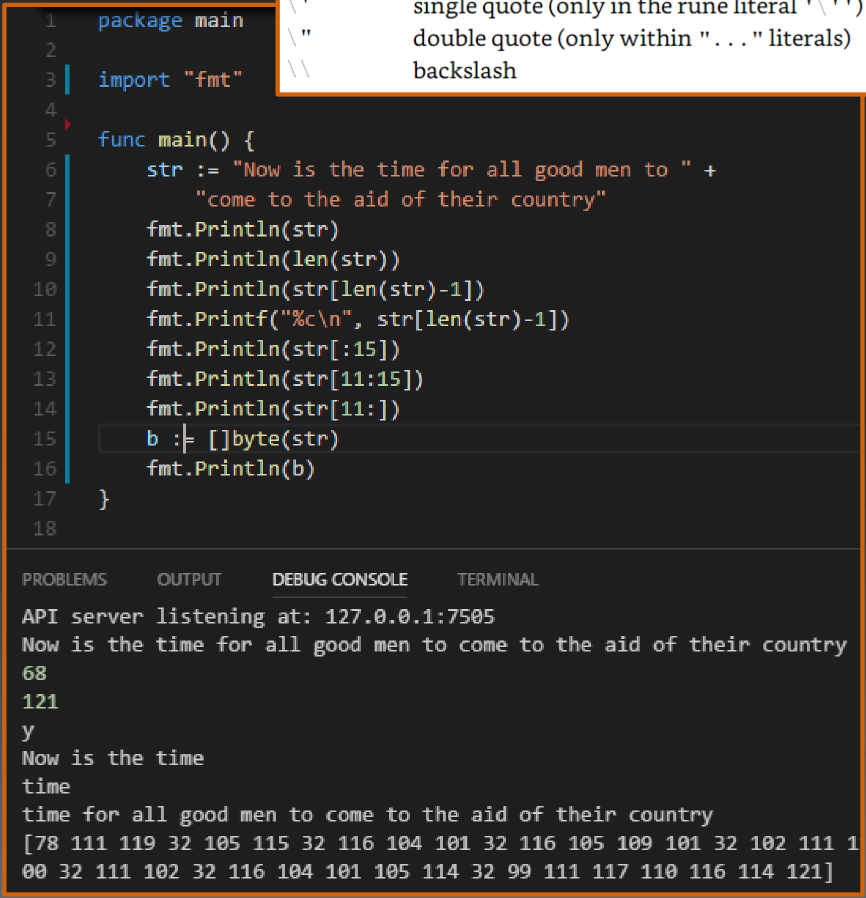
Strings

- A string is an immutable sequence of bytes
  - string
- Strings may contain arbitrary data, including bytes with value 0, but usually they contain human-readable text
- The len function returns the number of bytes (not runes) in a string
  - The i-th byte of a string is not necessarily the i-th character of a string because the UTF-8 encoding of a non-ASCII code point requires two or more bytes
- The index operation s[i] retrieves the i-th byte of string s
  - where 0 ≤ i < len(s)
  - Attempting to access a byte outside this range results in a panic
  - s[3] = ‘x’ //illegal, strings are immutable
  - s = “x” //ok, s gets new string (old string unch)
- The substring operation s[i:j] yields a new string consisting of the bytes of the original string starting at index i and continuing up to, but not including, the byte at index j
  - Omitting the i starts the substring at 0
  - Omitting the j stops the substring at len(s)
- Substrings share memory of the original (immutable) string
- Strings may be:
  - + Concatenated
  - += Concatenated and assigned back to the var
  - ==, <, <=, >, >=, != Compared lexically (byte by byte)
- String literals:
  - ‘x’ //Rune literal (not a string)
  - "Hello world\n" //Hello world with a new line (interpreted)
  - `Hello world\n` //Hello world\n (raw)
    - Good for json, html, regex, etc.



\a	“alert” or bell
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\'	single quote (only in the rune literal '\'')
\"	double quote (only within "..." literals)
\\	backslash

escape sequences

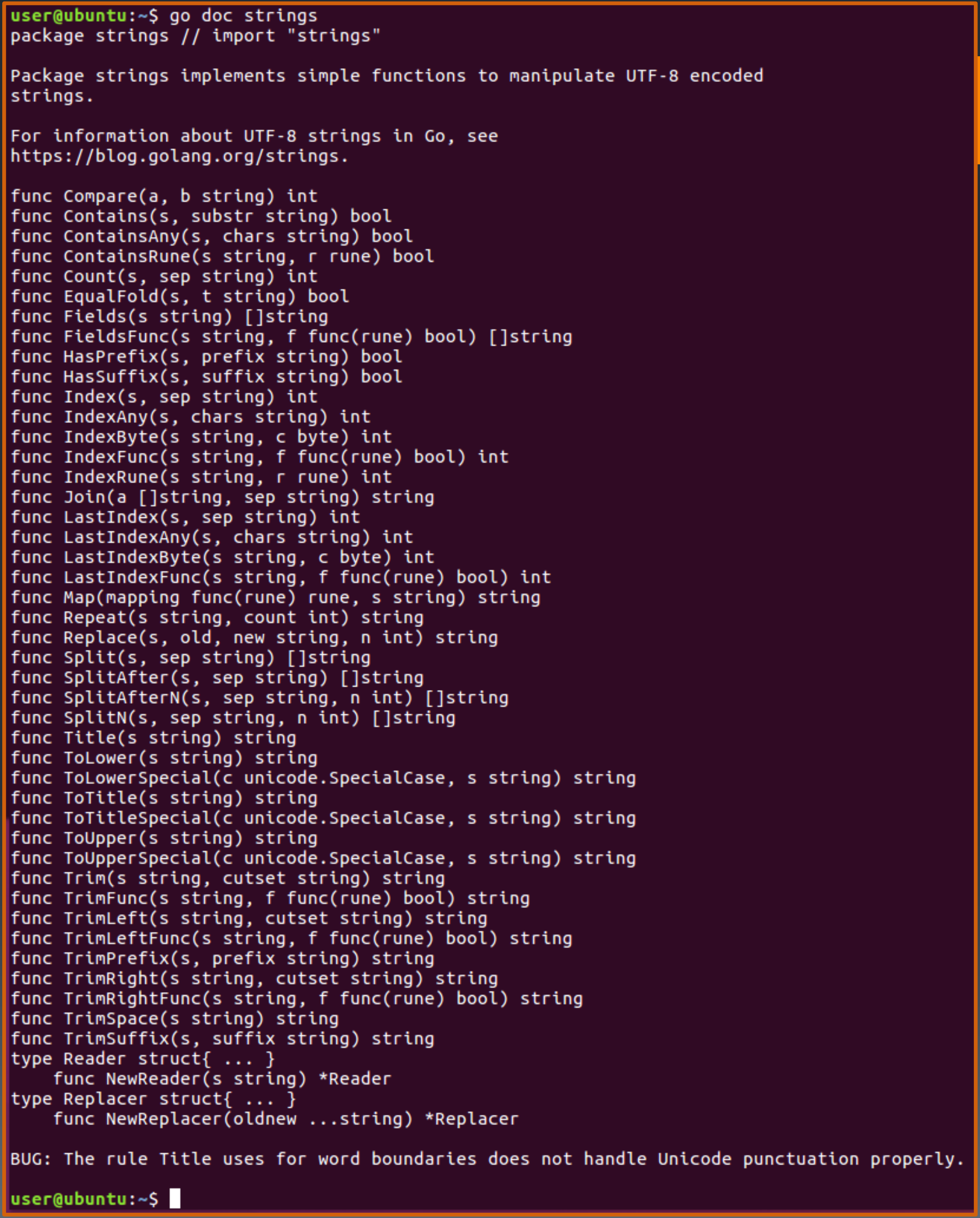


```go
package main

import "fmt"

func main() {
    str := "Now is the time for all good men to " +
        "come to the aid of their country"
    fmt.Println(str)
    fmt.Println(len(str))
    fmt.Println(str[len(str)-1])
    fmt.Printf("%c\n", str[len(str)-1])
    fmt.Println(str[:15])
    fmt.Println(str[11:15])
    fmt.Println(str[11:])
    b := []byte(str)
    fmt.Println(b)
}
```

```
API server listening at: 127.0.0.1:7505
Now is the time for all good men to come to the aid of their country
68
121
y
Now is the time
time
time for all good men to come to the aid of their country
[78 111 119 32 105 115 32 116 104 101 32 116 105 109 101 32 102 111 1
00 32 111 102 32 116 104 101 105 114 32 99 111 117 110 116 114 121]
```

String libs

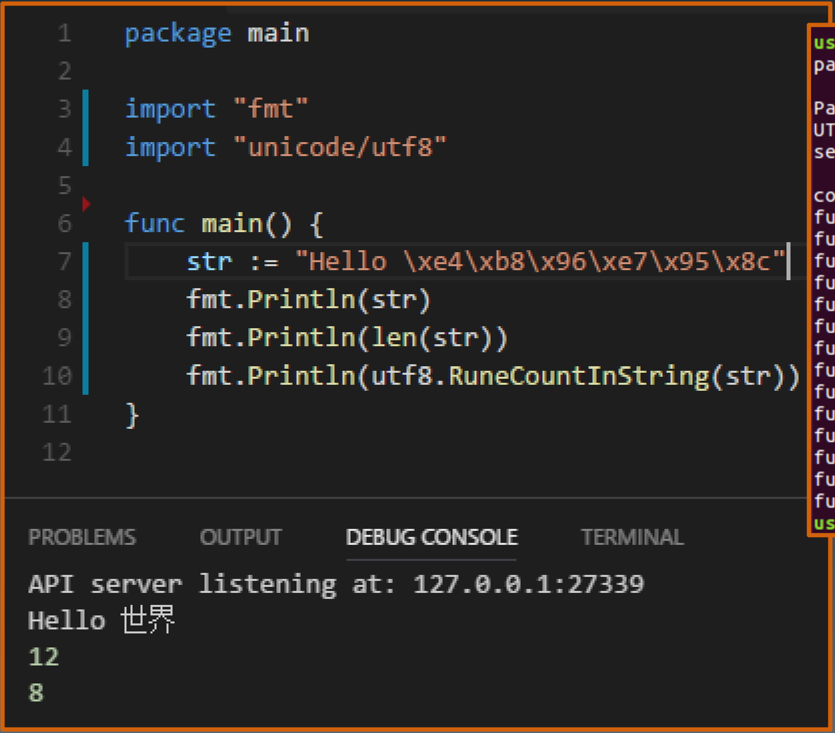
- Four standard packages are important for manipulating strings:
  - bytes
    - functions for manipulating slices of type [] byte
    - strings are immutable and building strings incrementally can be expensive
    - it's more efficient to use the bytes.Buffer
  - strings
    - functions for searching, replacing, comparing, trimming, splitting, and joining strings
  - strconv
    - functions for converting boolean, integer, and floating-point values to and from their string representations
    - functions for quoting and unquoting strings
  - unicode
    - functions like IsDigit, IsLetter, IsUpper, and IsLower for classifying runes



```
user@ubuntu:~$ go doc strings
package strings // import "strings"

Package strings implements simple functions to manipulate UTF-8 encoded
strings.

For information about UTF-8 strings in Go, see
https://blog.golang.org/strings.

func Compare(a, b string) int
func Contains(s, substr string) bool
func ContainsAny(s, chars string) bool
func ContainsRune(s string, r rune) bool
func Count(s, sep string) int
func EqualFold(s, t string) bool
func Fields(s string) []string
func FieldsFunc(s string, f func(rune) bool) []string
func HasPrefix(s, prefix string) bool
func HasSuffix(s, suffix string) bool
func Index(s, sep string) int
func IndexAny(s, chars string) int
func IndexByte(s string, c byte) int
func IndexFunc(s string, f func(rune) bool) int
func IndexRune(s string, r rune) int
func Join(a []string, sep string) string
func LastIndex(s, sep string) int
func LastIndexAny(s, chars string) int
func LastIndexByte(s string, c byte) int
func LastIndexFunc(s string, f func(rune) bool) int
func Map(mapping func(rune) rune, s string) string
func Repeat(s string, count int) string
func Replace(s, old, new string, n int) string
func Split(s, sep string) []string
func SplitAfter(s, sep string) []string
func SplitAfterN(s, sep string, n int) []string
func SplitN(s, sep string, n int) []string
func Title(s string) string
func ToLower(s string) string
func ToLowerSpecial(c unicode.SpecialCase, s string) string
func ToTitle(s string) string
func ToTitleSpecial(c unicode.SpecialCase, s string) string
func ToUpper(s string) string
func ToUpperSpecial(c unicode.SpecialCase, s string) string
func Trim(s string, cutset string) string
func TrimFunc(s string, f func(rune) bool) string
func TrimLeft(s string, cutset string) string
func TrimLeftFunc(s string, f func(rune) bool) string
func TrimPrefix(s, prefix string) string
func TrimRight(s string, cutset string) string
func TrimRightFunc(s string, f func(rune) bool) string
func TrimSpace(s string) string
func TrimSuffix(s, suffix string) string
type Reader struct{ ... }
    func NewReader(s string) *Reader
type Replacer struct{ ... }
    func NewReplacer(oldnew ...string) *Replacer

BUG: The rule Title uses for word boundaries does not handle Unicode punctuation properly.

user@ubuntu:~$ 
```

Unicode

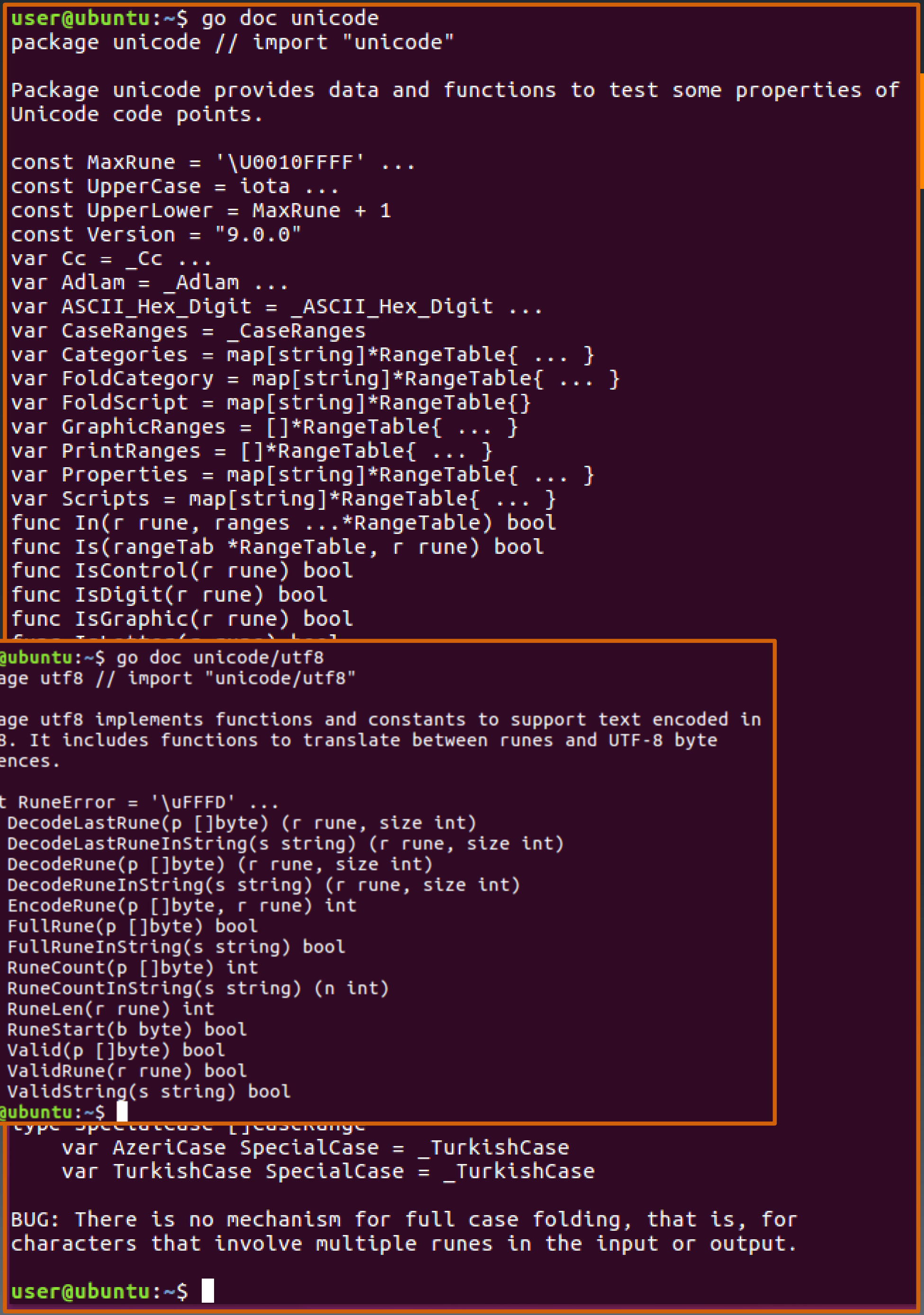
- Text strings are conventionally interpreted as UTF-8-encoded sequences of Unicode code points (runes)
  - UTF-8 was invented by Ken Thompson and Rob Pike, two of the creators of Go
- Go source files are always in UTF-8



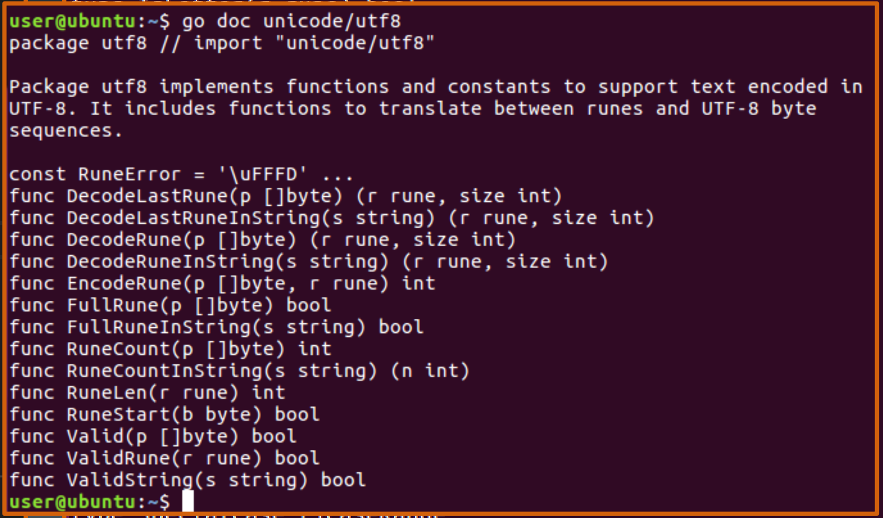
```go
package main

import "fmt"
import "unicode/utf8"

func main() {
    str := "Hello \xe4\xb8\x96\xe7\x95\x8c"
    fmt.Println(str)
    fmt.Println(len(str))
    fmt.Println(utf8.RuneCountInString(str))
}
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
API server listening at: 127.0.0.1:27339
Hello 世界
12
8
```



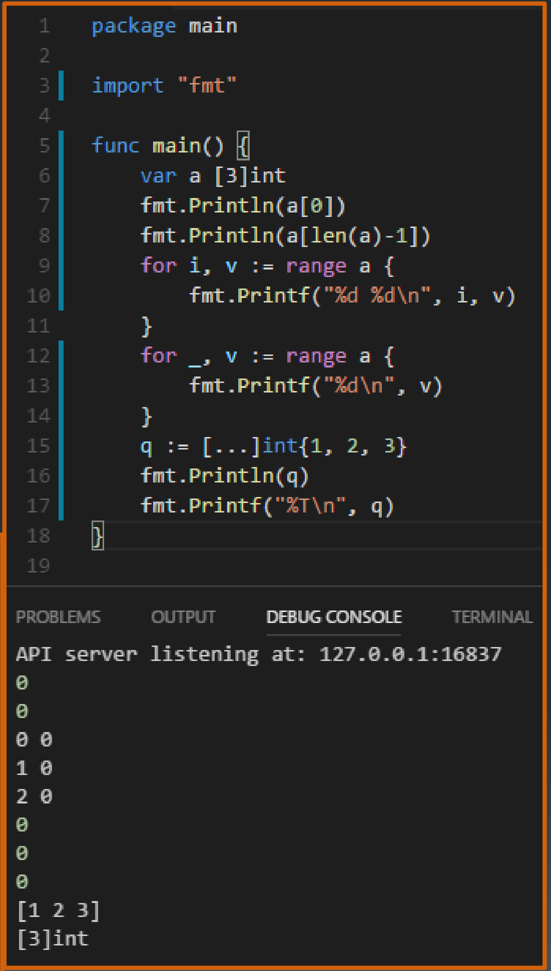
```
user@ubuntu:~$ go doc unicode
package unicode // import "unicode"

Package unicode provides data and functions to test some properties of
Unicode code points.

const MaxRune = '\U0010FFFF' ...
const UpperCase = iota ...
const UpperLower = MaxRune + 1
const Version = "9.0.0"
var Cc = _Cc ...
var Adlam = _Adlam ...
var ASCII_Hex_Digit = _ASCII_Hex_Digit ...
var CaseRanges = _CaseRanges
var Categories = map[string]*RangeTable{ ... }
var FoldCategory = map[string]*RangeTable{ ... }
var FoldScript = map[string]*RangeTable{}
var GraphicRanges = []*RangeTable{ ... }
var PrintRanges = []*RangeTable{ ... }
var Properties = map[string]*RangeTable{ ... }
var Scripts = map[string]*RangeTable{ ... }
func In(r rune, ranges ...*RangeTable) bool
func Is(rangeTab *RangeTable, r rune) bool
func IsControl(r rune) bool
func IsDigit(r rune) bool
func IsGraphic(r rune) bool

    var AzeriCase SpecialCase = _TurkishCase
    var TurkishCase SpecialCase = _TurkishCase

BUG: There is no mechanism for full case folding, that is, for
characters that involve multiple runes in the input or output.

user@ubuntu:~$
```



```
user@ubuntu:~$ go doc unicode/utf8
package utf8 // import "unicode/utf8"

Package utf8 implements functions and constants to support text encoded in
UTF-8. It includes functions to translate between runes and UTF-8 byte
sequences.

const RuneError = '\uFFFD' ...
func DecodeLastRune(p []byte) (r rune, size int)
func DecodeLastRuneInString(s string) (r rune, size int)
func DecodeRune(p []byte) (r rune, size int)
func DecodeRuneInString(s string) (r rune, size int)
func EncodeRune(p []byte, r rune) int
func FullRune(p []byte) bool
func FullRuneInString(s string) bool
func RuneCount(p []byte) int
func RuneCountInString(s string) (n int)
func RuneLen(r rune) int
func RuneStart(b byte) bool
func Valid(p []byte) bool
func ValidRune(r rune) bool
func ValidString(s string) bool
user@ubuntu:~$
```

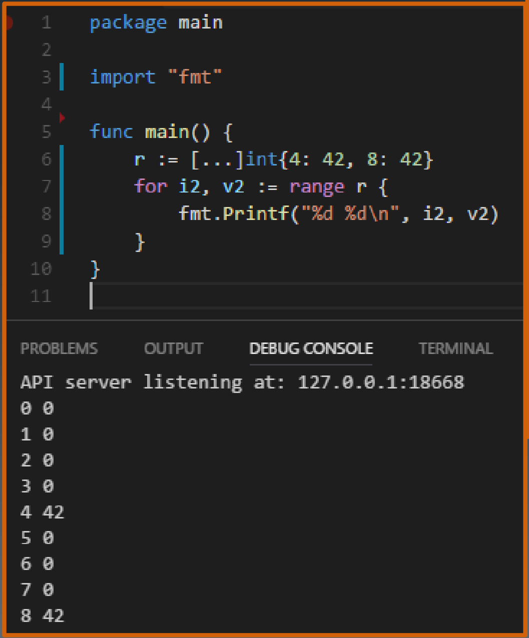
Arrays

- An array is a fixed-length sequence of zero or more elements of a particular type
- Because of their fixed length, arrays are rarely used directly in Go
  - Slices, which can grow and shrink, are much more versatile
- Individual array elements are accessed with the conventional subscript notation x[i]
  - Subscripts run from zero to one less than the array length
- The built-in function len returns the number of elements in the array
- Array literals are enclosed in curly braces
  - 2 elements: {2, 4}
  - Sparse init: {3: 1.1, 5: 1.2}
- Ellipsis can be used to infer array size […]
  - The %T conversion displays an object's type



```go
package main

import "fmt"

func main() {
    var a [3]int
    fmt.Println(a[0])
    fmt.Println(a[len(a)-1])
    for i, v := range a {
        fmt.Printf("%d %d\n", i, v)
    }
    for _, v := range a {
        fmt.Printf("%d\n", v)
    }
    q := [...]int{1, 2, 3}
    fmt.Println(q)
    fmt.Printf("%T\n", q)
}
```

```
API server listening at: 127.0.0.1:16837
0
0
0 0
1 0
2 0
0
0
0
[1 2 3]
[3]int
```

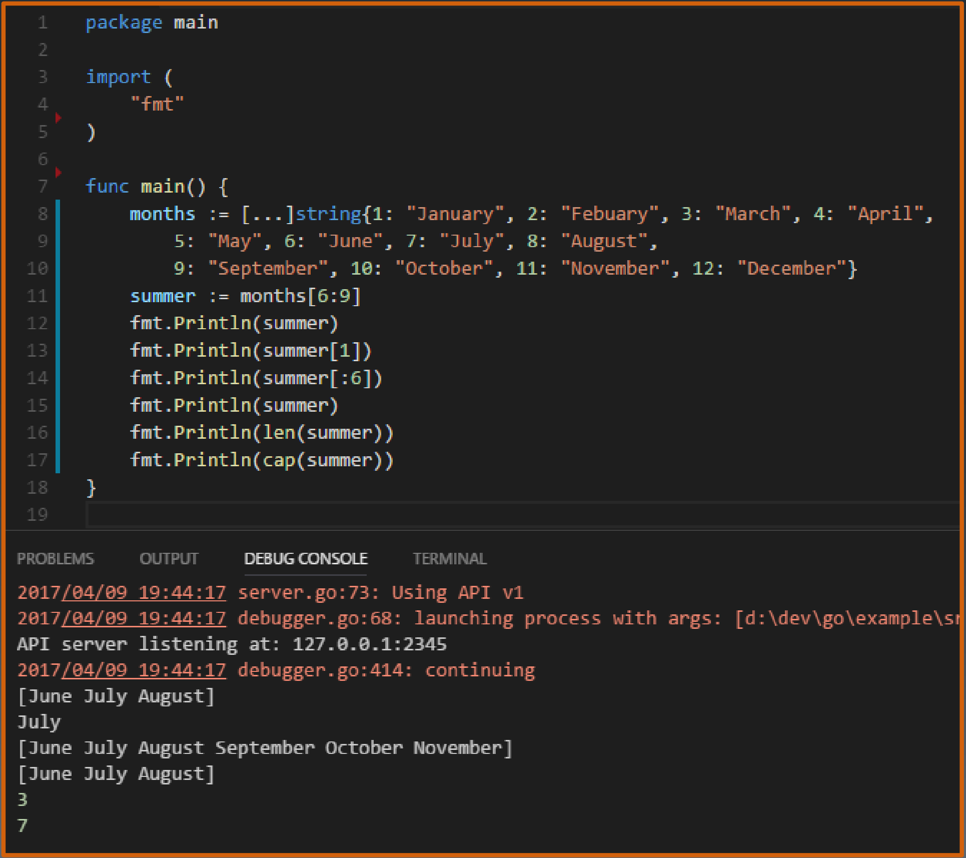


```go
package main

import "fmt"

func main() {
    r := [...]int{4: 42, 8: 42}
    for i2, v2 := range r {
        fmt.Printf("%d %d\n", i2, v2)
    }
}
```

```
API server listening at: 127.0.0.1:18668
0 0
1 0
2 0
3 0
4 42
5 0
6 0
7 0
8 42
```

Slices

- A slice is a lightweight data structure that gives access to a subsequence (or perhaps all) of the elements of an array
  - known as the slice's underlying array
- Slices represent variable-length sequences whose elements all have the same type
- A slice type is written []T, where the elements have type T
  - Looks like an array type without a size
- A slice has three components:
  - A pointer
    - To the first element of the array reachable through the slice
  - A length
    - The number of slice elements (can't exceed the capacity)
    - Returned by the built-in function len()
  - A capacity
    - The number of elements between the start of the slice and the end of the underlying array
    - Returned by the built-in function cap()
- The built-in append function appends items to slices
  - summer = append(summer, "Freemonth")
  - Modifies underlying array!
- Unlike arrays, slices are not comparable
  - Cannot use == to test two slices for the same elements
- The slice zero value is nil

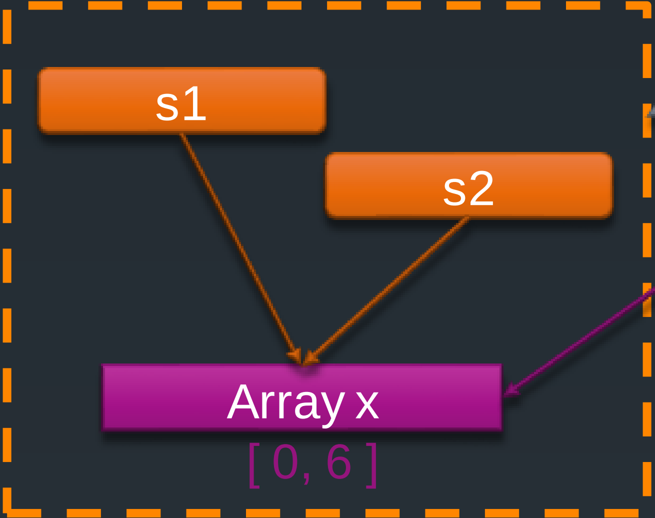


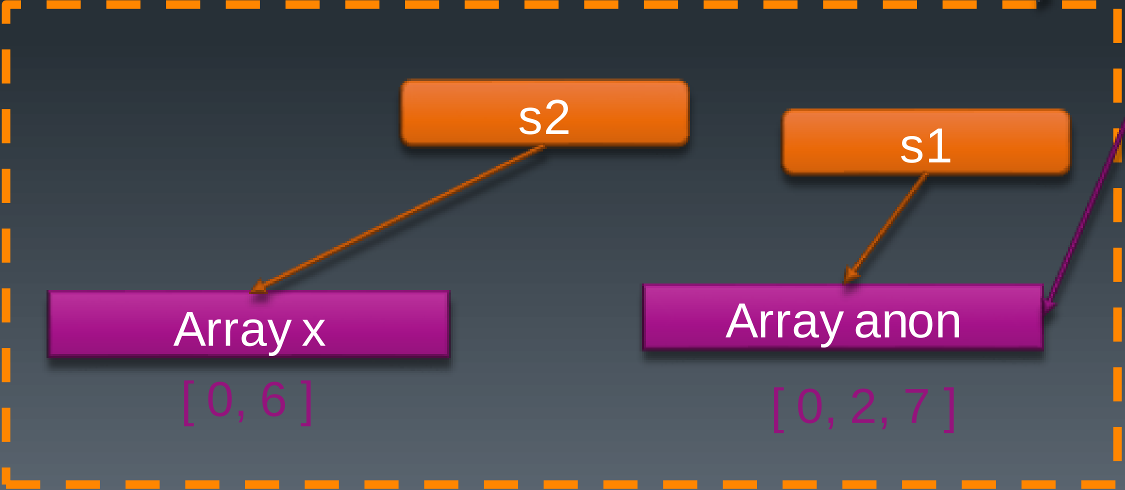
```go
package main

import (
	"fmt"
)

func main() {
	months := [...]string{1: "January", 2: "Febuary", 3: "March", 4: "April",
		5: "May", 6: "June", 7: "July", 8: "August",
		9: "September", 10: "October", 11: "November", 12: "December"}
	summer := months[6:9]
	fmt.Println(summer)
	fmt.Println(summer[1])
	fmt.Println(summer[:6])
	fmt.Println(summer)
	fmt.Println(len(summer))
	fmt.Println(cap(summer))
}
```

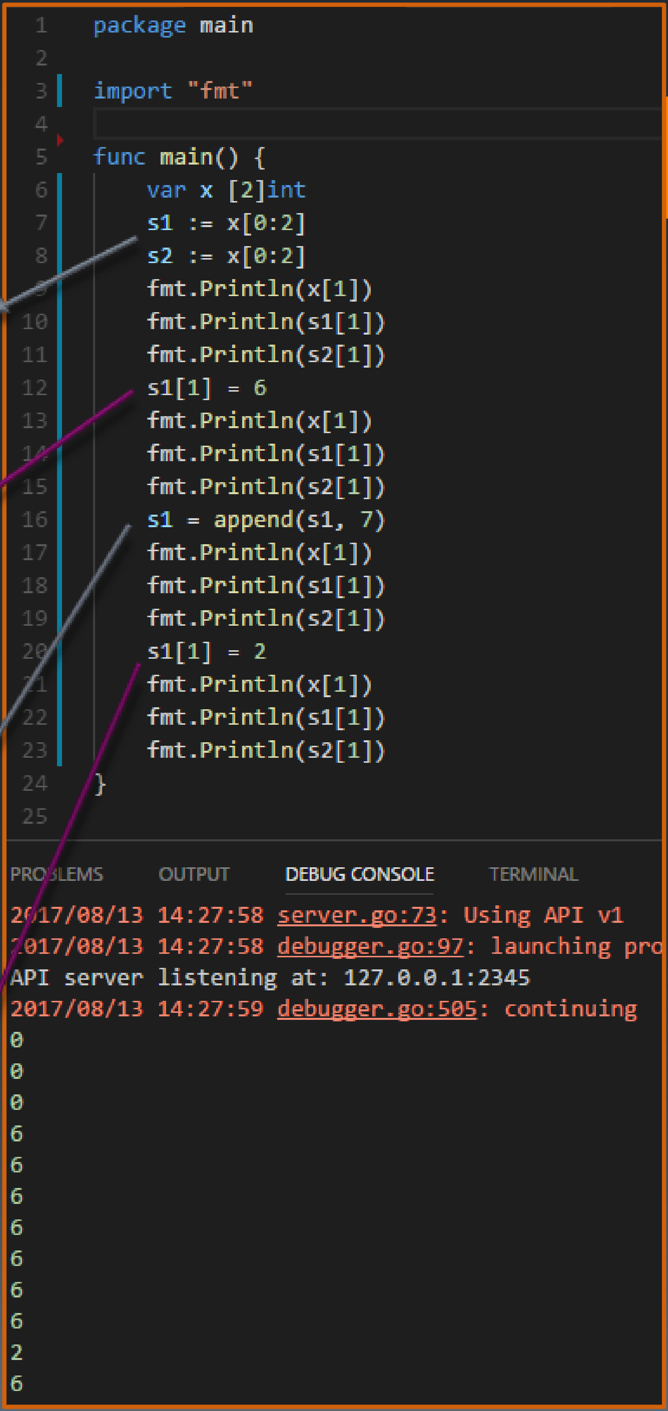
```
2017/04/09 19:44:17 server.go:73: Using API v1
2017/04/09 19:44:17 debugger.go:68: launching process with args: [d:\dev\go\example\s
API server listening at: 127.0.0.1:2345
2017/04/09 19:44:17 debugger.go:414: continuing
[June July August]
July
[June July August September October November]
[June July August]
3
7
```

Append Internals

- Appending to a slice:
  - Increases the slice's length
  - If (and only if) the length exceeds the capacity of the underlying array the underlying array is copied to a new anonymous array with the necessary length and the slice capacity is updated to the same value as the length







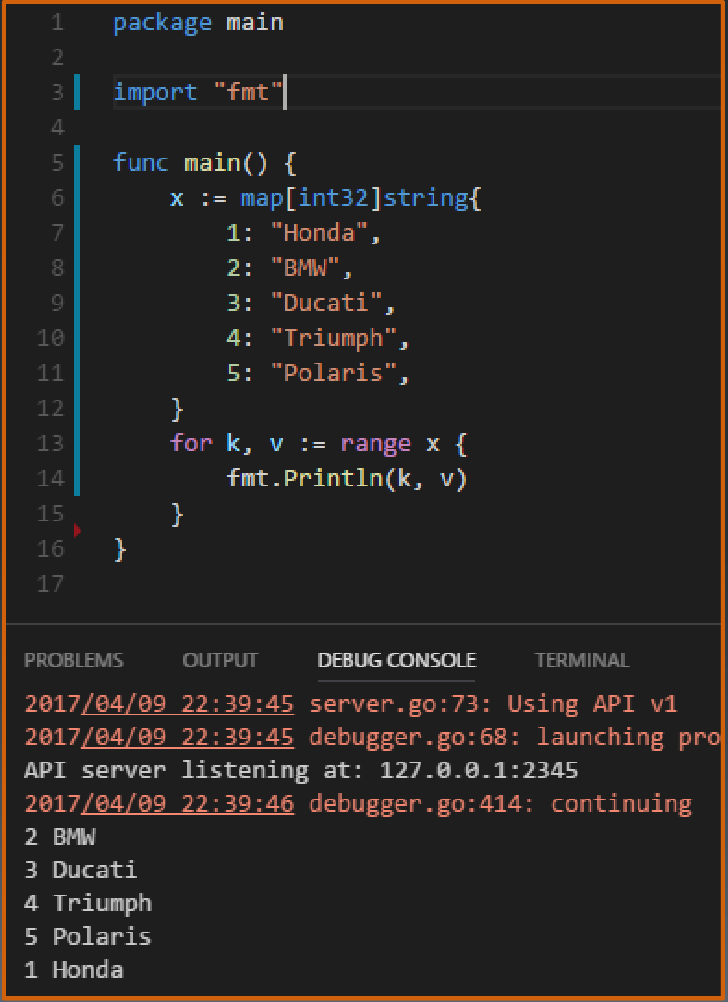
```go
package main

import "fmt"

func main() {
	var x [2]int
	s1 := x[0:2]
	s2 := x[0:2]
	fmt.Println(x[1])
	fmt.Println(s1[1])
	fmt.Println(s2[1])
	s1[1] = 6
	fmt.Println(x[1])
	fmt.Println(s1[1])
	fmt.Println(s2[1])
	s1 = append(s1, 7)
	fmt.Println(x[1])
	fmt.Println(s1[1])
	fmt.Println(s2[1])
	s1[1] = 2
	fmt.Println(x[1])
	fmt.Println(s1[1])
	fmt.Println(s2[1])
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/08/13 14:27:58 server.go:73: Using API v1
2017/08/13 14:27:58 debugger.go:97: launching pro
API server listening at: 127.0.0.1:2345
2017/08/13 14:27:59 debugger.go:505: continuing
0
0
0
6
6
6
6
6
6
6
2
6
```

Maps

- A map is an unordered collection of key/value pairs
  - All keys are distinct
  - Value associated with a given key can be retrieved, updated, or removed
  - All operations are constant time O(1)
- Map types are written map[K] V
  - K is the key type
    - The key type K must be comparable using ==
    - Floating-point keys are almost always a bad idea as they are hard to compare for equality
  - V is the value type
- The built-in make function make can be used to create a map
  - motorbike := make(map[int32] string)
- Map literals are supported
  - motorbike := map[int32] string {
    1: “Honda”,
    2: “BMW”,
    }
- Maps are accessed through the hash key
  - var bike string = motorbike[1]
- The built in delete function removes elements
  - delete(motorbike, 1)
- Unknown keys return the zero value for the stored type in the above operations
- You can not take the address of a map element
- You can iterate over a map with a range based for loop
  - The order is however unspecified



```go
package main

import "fmt"

func main() {
    x := map[int32]string{
        1: "Honda",
        2: "BMW",
        3: "Ducati",
        4: "Triumph",
        5: "Polaris",
    }
    for k, v := range x {
        fmt.Println(k, v)
    }
}
```

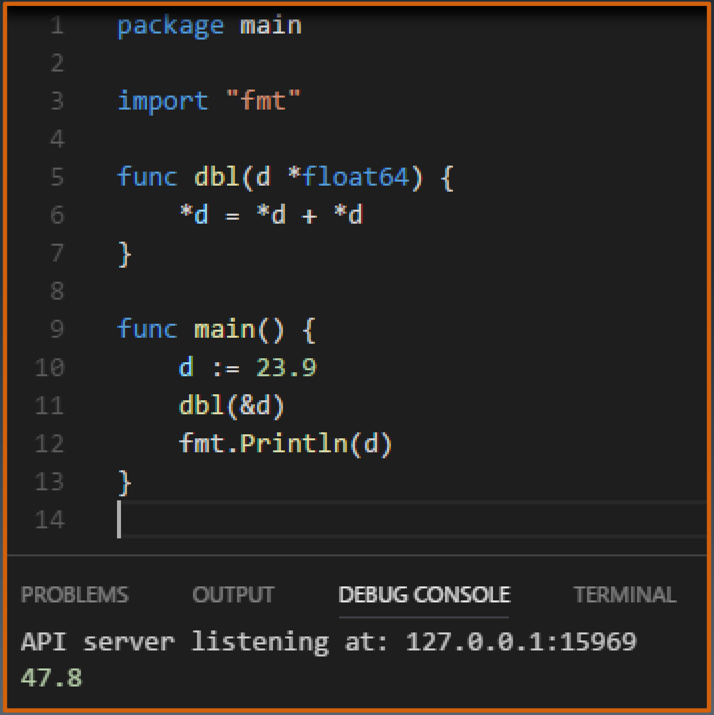
```
2017/04/09 22:39:45 server.go:73: Using API v1
2017/04/09 22:39:45 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/09 22:39:46 debugger.go:414: continuing
2 BMW
3 Ducati
4 Triumph
5 Polaris
1 Honda
```

Sets

- Go does not provide a set type
- However a map's keys are a set
- Defining a set of strings with a small ignored value type is common in go:
  - myset := make(map [string] bool)

Pointers

- A pointer value is the address of a variable
  - var x int
  - &x (address of x) yields a pointer to the integer variable
  - *int is type “pointer to int”
  - *p yields the value of the pointed to variable
- Every variable has an address
- The zero value for pointers is nil
- Pointers are equal only if they are both nil or they both point to the same variable
- In Go returning a pointer to a local variable is safe
  - Stop hyperventilating C++ programmers
  - Go maintains the memory as long as there are references to it

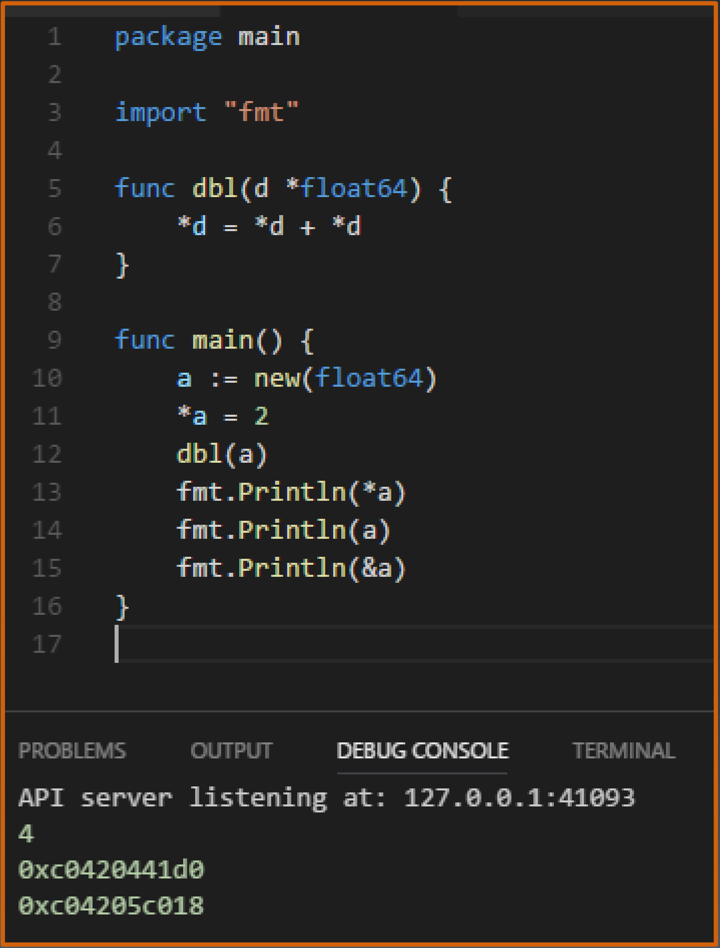


```go
package main

import "fmt"

func dbl(d *float64) {
	*d = *d + *d
}

func main() {
	d := 23.9
	dbl(&d)
	fmt.Println(d)
}

```

```
API server listening at: 127.0.0.1:15969
47.8
```

New

- The built in new(type) function creates a new unnamed pointer variable and initializes it to the zero value for the specified type
- New is a built in function and can be masked by locals using the name “new”
  - It should be clear that this is a fairly bad idea

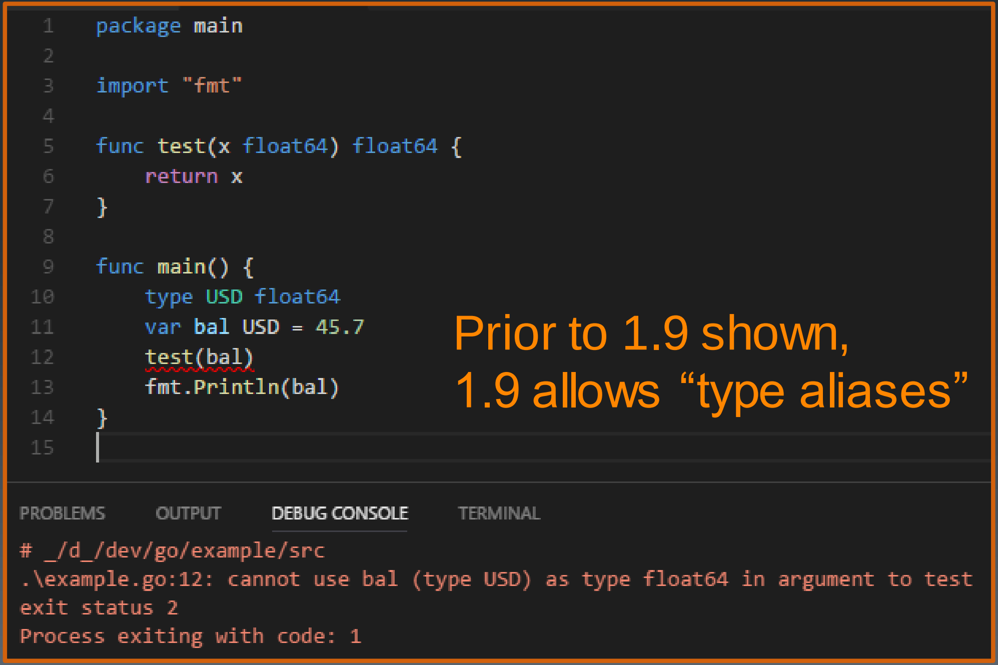


```go
package main

import "fmt"

func dbl(d *float64) {
	*d = *d + *d
}

func main() {
	a := new(float64)
	*a = 2
	dbl(a)
	fmt.Println(*a)
	fmt.Println(a)
	fmt.Println(&a)
}

```

```
API server listening at: 127.0.0.1:41093
4
0xc0420441d0
0xc04205c018
```

Type Declarations

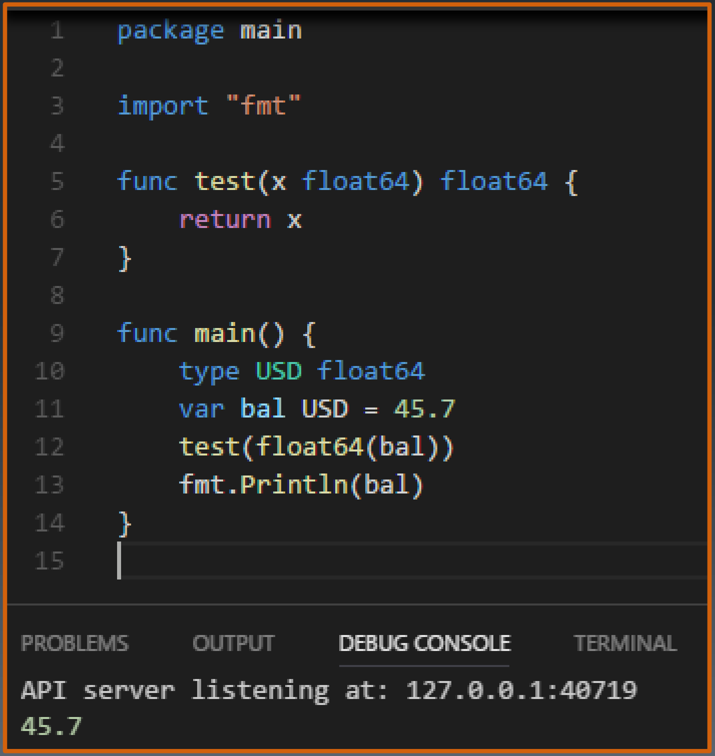
- A type declaration defines a new named type that has the same underlying type as an existing type
- Typically used at the package level for internal package types
- For every type T, there is a corresponding conversion operation T(x) that converts the value x to type T
  - A conversion from one type to another is allowed if both have the same underlying type



```go
package main

import "fmt"

func test(x float64) float64 {
	return x
}

func main() {
	type USD float64
	var bal USD = 45.7
	test(bal)
	fmt.Println(bal)
}

```

```
# _/d_/dev/go/example/src
.\example.go:12: cannot use bal (type USD) as type float64 in argument to test
exit status 2
Process exiting with code: 1
```

Prior to 1.9 shown,
1.9 allows "type aliases"



```go
package main

import "fmt"

func test(x float64) float64 {
	return x
}

func main() {
	type USD float64
	var bal USD = 45.7
	test(float64(bal))
	fmt.Println(bal)
}

```

```
API server listening at: 127.0.0.1:40719
45.7
```

Summary

- Concepts covered:
  - Integers
  - Floats
  - Strings
  - Key type manipulation packages
  - Slices
  - Arrays
  - Maps

Lab: Data Types and Composite Types

- Working with Types