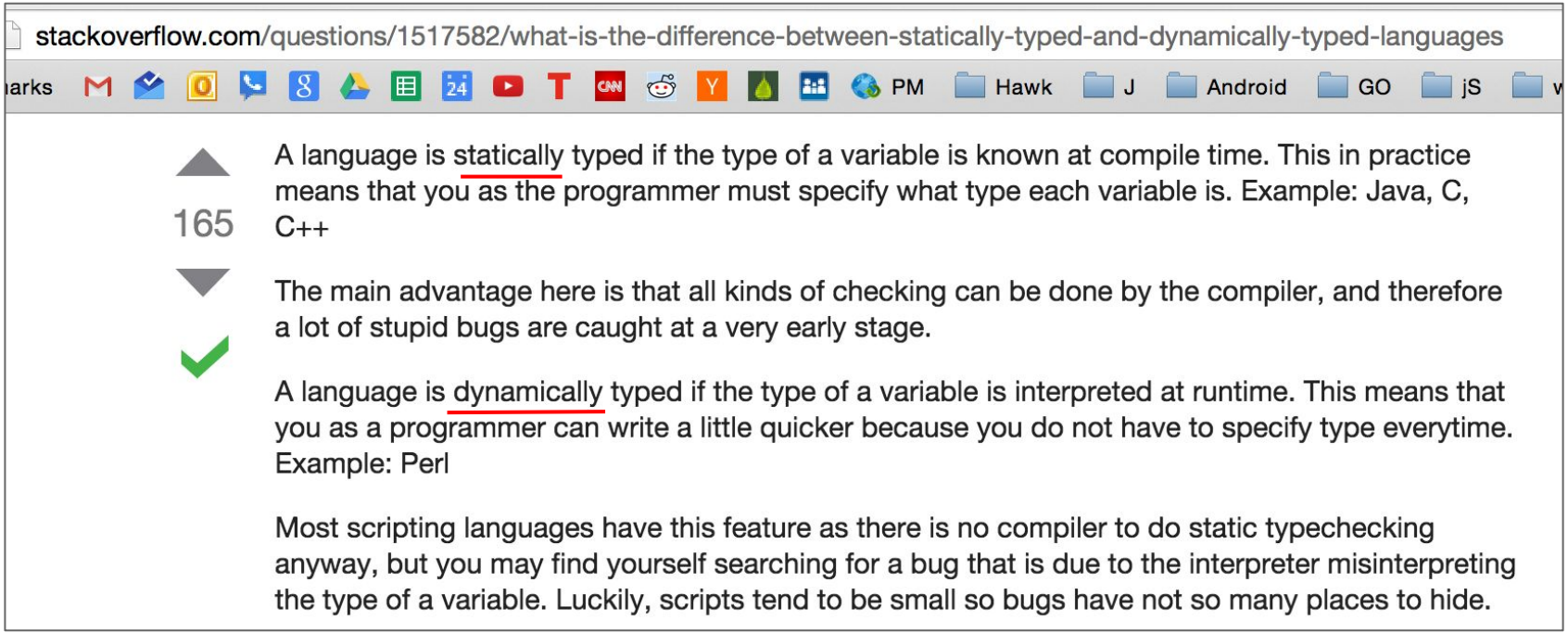
Types

int, float, string, rune, bool
len, concat, strconv, conversion, assertion
math, rand, docs

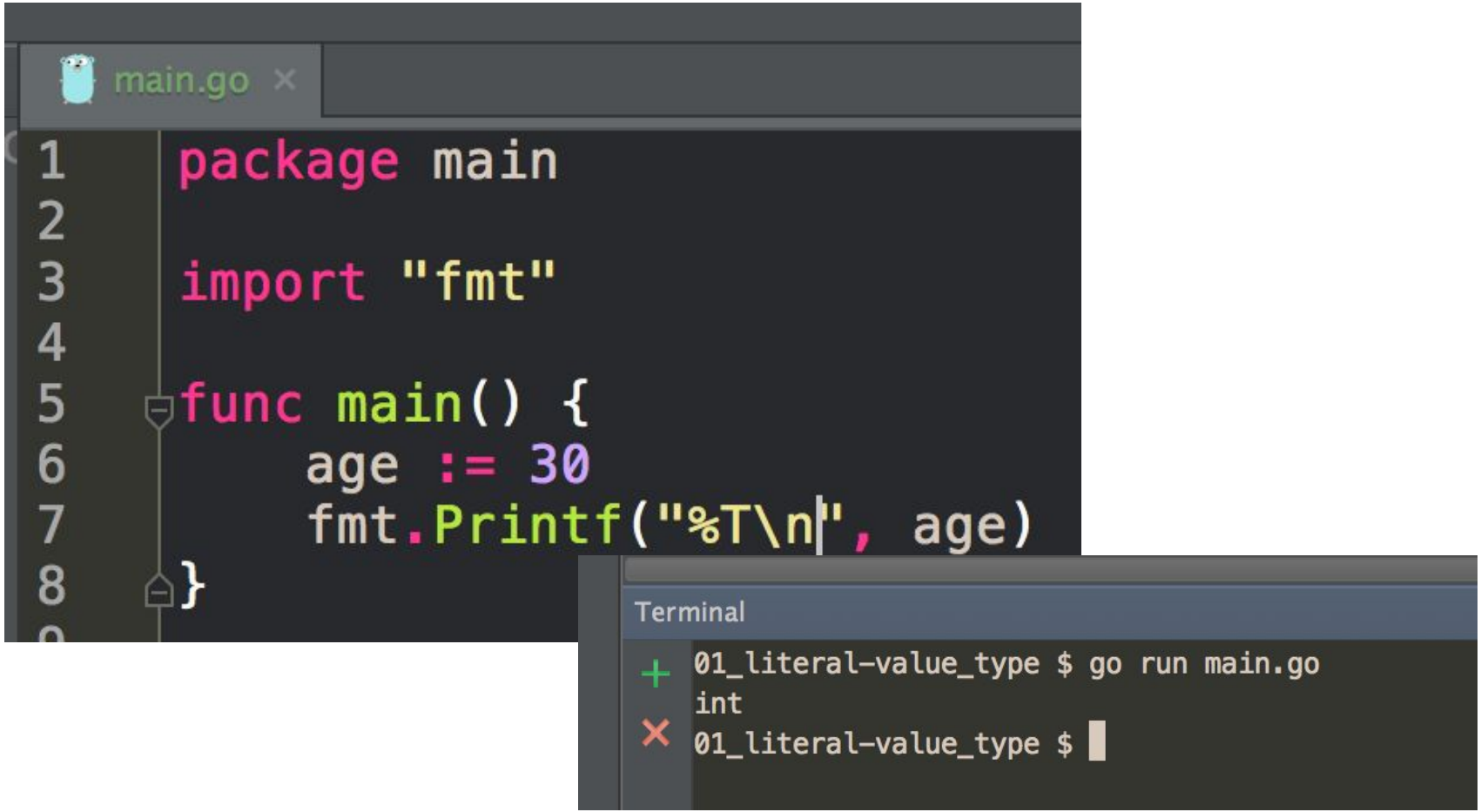
statically typed

types

- **statically typed**
  - languages like javascript are dynamically typed
  - this a big difference between golang (statically typed) and other languages that are dynamically typed
  - strong typing / static typing helps prevent errors


stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages
arks
PM
Hawk
J
Android
GO
jS
165
A language is statically typed if the type of a variable is known at compile time. This in practice means that you as the programmer must specify what type each variable is. Example: Java, C, C++
The main advantage here is that all kinds of checking can be done by the compiler, and therefore a lot of stupid bugs are caught at a very early stage.
A language is dynamically typed if the type of a variable is interpreted at runtime. This means that you as a programmer can write a little quicker because you do not have to specify type everytime. Example: Perl
Most scripting languages have this feature as there is no compiler to do static typechecking anyway, but you may find yourself searching for a bug that is due to the interpreter misinterpreting the type of a variable. Luckily, scripts tend to be small so bugs have not so many places to hide.

viewing type


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 age := 30
7 fmt.Printf("%T\n", age)
8 }
Terminal
01_literal-value_type $ go run main.go
int
01_literal-value_type $

main.go

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    age := 30
    fmt.Printf("%T\n", age)
    fmt.Println(reflect.TypeOf(age))
}

```

Terminal

```
02_literal-value_TypeOf $ go run main.go
int
int
02_literal-value_TypeOf $
```

main.go

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    age := 30.74
    fmt.Printf("%T\n", age)
    fmt.Println(reflect.TypeOf(age))
}
```

Terminal

```
03_literal-value_TypeOf_float $ go run main.go
float64
float64
03_literal-value_TypeOf_float $
```

main.go

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    fname := "Jane"
    lname := `Doe`
    fmt.Printf("%T\n", fname)
    fmt.Println(reflect.TypeOf(lname))
}

```

Terminal

```
03_literal-value_TypeOf $ go run main.go
string
string
03_literal-value_TypeOf $
```

question

what does this represent:
00001010

Answer

- we don’t know until we know the type:
  - is it base 2 int?
  - is it a color?
  - is it a code point?

type: size & representation

Type

- Provides the compiler with two pieces of information:
  - **size**
  - **representation**

Type

- Provides the compiler with two pieces of information:
  - **size**
    - how much memory to allocate
  - **representation**
    - what the memory represents

float64

gives you two pieces of information

- **size**: 64 bits
- **representation**: float

int

architecture specific

- **size**: “word” size - 4 or 8 bytes (eg, 32 or 64 bits)
- **representation**: int

[256]int

- **size**: 256 bytes
- **representation**: int

[256]bool

- **size**: 256 bytes
- **representation**: bool

[256]byte

- **size**: 256 bytes
- **representation**: byte

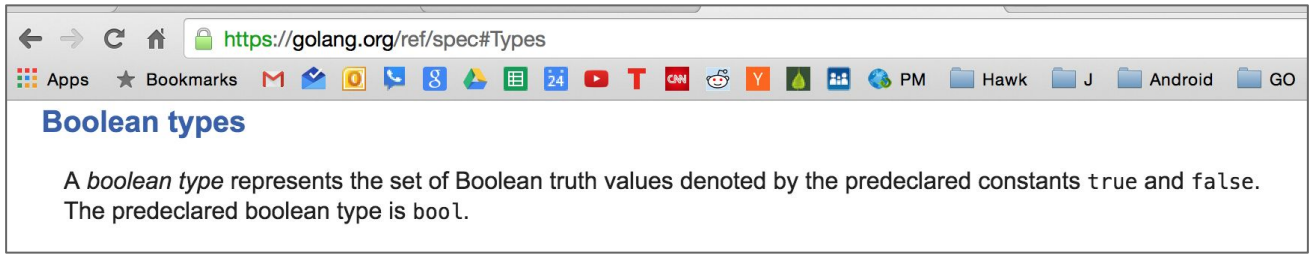
language spec

Types

- Method sets
- Boolean types
- Numeric types
- String types
- Array types
- Slice types
- Struct types
- Pointer types
- Function types
- Interface types
- Map types
- Channel types

https://golang.org/ref/spec

boolean


https://golang.org/ref/spec#Types
Apps
Bookmarks
PM
Hawk
J
Android
GO
Boolean types
A boolean type represents the set of Boolean truth values denoted by the predeclared constants true and false.
The predeclared boolean type is bool.

main.go

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    content := true
    fmt.Printf("%T\n", content)
    fmt.Println(reflect.TypeOf(content))
}
```

Terminal

```
06_literal-value_TypeOf_bool $ go run main.go
bool
bool
06_literal-value_TypeOf_bool $
```

numeric

https://golang.org/ref/spec#Numeric_types

Apps Bookmarks PM Hawk J Androi

Numeric types

A *numeric type* represents sets of integer or floating-point values. The predeclared architecture-independent numeric types are:

```
uint8       the set of all unsigned  8-bit integers (0 to 255)
uint16      the set of all unsigned 16-bit integers (0 to 65535)
uint32      the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8        the set of all signed  8-bit integers (-128 to 127)
int16       the set of all signed 16-bit integers (-32768 to 32767)
int32       the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64       the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

float32     the set of all IEEE-754 32-bit floating-point numbers
float64     the set of all IEEE-754 64-bit floating-point numbers

complex64   the set of all complex numbers with float32 real and imaginary parts
complex128  the set of all complex numbers with float64 real and imaginary parts

byte        alias for uint8
rune        alias for int32
```

u - unsigned

- meaning, no negative (-) values, only positive (+) values
- eg, it has no sign to say positive/negative, they're all just positive;
- that's why you have twice as many as signed

The value of an *n*-bit integer is *n* bits wide and represented using two's complement arithmetic.

There is also a set of predeclared numeric types with implementation-specific sizes:

determined by your cpu word size: 32bit or 64bit

```
uint     either 32 or 64 bits
int      same size as uint
uintptr  an unsigned integer large enough to store the uninterpreted bits of a pointer value
```

use "int" unless you have hardware reasons to use "int8" or "int16"

To avoid portability issues all numeric types are distinct except `byte`, which is an alias for `uint8`, and `rune`, which is an alias for `int32`. Conversions are required when different numeric types are mixed in an expression or assignment. For instance, `int32` and `int` are not the same type even though they may have the same size on a particular architecture.

https://golang.org/ref/spec#Numeric_types

Apps Bookmarks Hawk J Androi

Numeric types

A *numeric type* represents sets of integer or floating-point values. The pre dent
numeric types are:

```
uint8       the set of all unsigned  8-bit integers (
uint16      the set of all unsigned 16-bit integers
uint32      the set of all unsigned 32-bit inte
uint64      the set of all unsigned 64-bit int

int8        the set of all signed  8-bit i
int16       the set of all signed 16-bi
int32       the set of all signed 32-
int64       the set of all signed 6                                    036854775807)

float32     the set of all IEE
float64     the set of all I

complex64   the set of
complex128  the set o                                                  ary parts
                                                                       ginary parts

byte        alias
rune        ali
```

The value o o's complement arithmetic.

There itation-specific sizes:

```
uint
int
uintptr                                  to store the uninterpreted bits of a pointer value
```

To avoid portability are distinct except `byte`, which is an alias for `uint8`, and `rune`, which is an alias for `int32`. C quired when different numeric types are mixed in an expression or assignment. For instance, `int` are not the same type even though they may have the same size on a particular architecture.

"Word size" refers to the number of bits processed by a computer's CPU in one go (these days, typically **32 bits** or **64 bits**). Data bus size, instruction size, address size are usually multiples of the word size. Nov 6, 2013

what does it mean by word size in computer? - Stack Overflow
stackoverflow.com/questions/.../what-does-it-mean-by-**word-size**-in-compu...

igned
meaning, no negative (-) values, only positive
values
it has no sign to say positive/negative,
hey're all just positive;
that's why you have twice as many as signed

determined by your cpu word size: 32bit or 64bit

main.go

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    rune := 'd'
    fmt.Printf("%T\n", rune)
    fmt.Println(reflect.TypeOf(rune))
}
```

Terminal

```
04_literal-value_TypeOf $ go run main.go
int32
int32
04_literal-value_TypeOf $
```

rune
/ro͞on/

noun

a letter of an ancient Germanic alphabet, related to the Roman alphabet.

- a mark or letter of mysterious or magic significance.
- small stones, pieces of bone, etc., bearing runes, and used as divinatory symbols.
  "the casting of the runes"

https://golang.org/ref/spec#Rune_literals

Apps Bookmarks PM Hawk J Android

Rune literals

A rune literal represents a rune constant, an integer value identifying a Unicode code point. A rune literal is expressed as one or more characters enclosed in single quotes, as in `'x'` or `'\n'`. Within the quotes, any character may appear except newline and unescaped single quote. A single quoted character represents the Unicode value of the character itself, while multi-character sequences beginning with a backslash encode values in various formats.

https://golang.org/ref/spec#Numeric_types

Apps Bookmarks

```
byte        alias for uint8
rune        alias for int32
```

main.go

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    rune := 'd'
    fmt.Printf("%T\n", rune)
    fmt.Println(reflect.TypeOf(rune))
}
```

Terminal

```
04_literal-value_TypeOf $ go run main.go
int32
int32
04_literal-value_TypeOf $
```

rune
/roon/

noun

a letter of an ancient Germanic alphabet, related to the Roman alphabet.

- a mark or letter of mysterious or magic significance.
- small stones, pieces of bone, etc., bearing runes, and used as divinatory symbols.
  "the casting of the runes"

https://golang.org/ref/spec#Rune_literals

Apps Bookmarks PM Hawk J Android

Rune literals

A rune literal represents a rune constant, an integer value identifying a Unicode code point. A rune literal is expressed as one or more characters enclosed in single quotes, as in `'x'` or `'\n'`. Within the quotes, any character may appear except newline and unescaped single quote. A single quoted character represents the Unicode value of the character itself, while multi-character sequences beginning with a backslash encode values in various formats.

https://golang.org/ref/spec#Numeric_types

Apps Bookmarks

```
byte        alias for uint8
rune        alias for int32
```

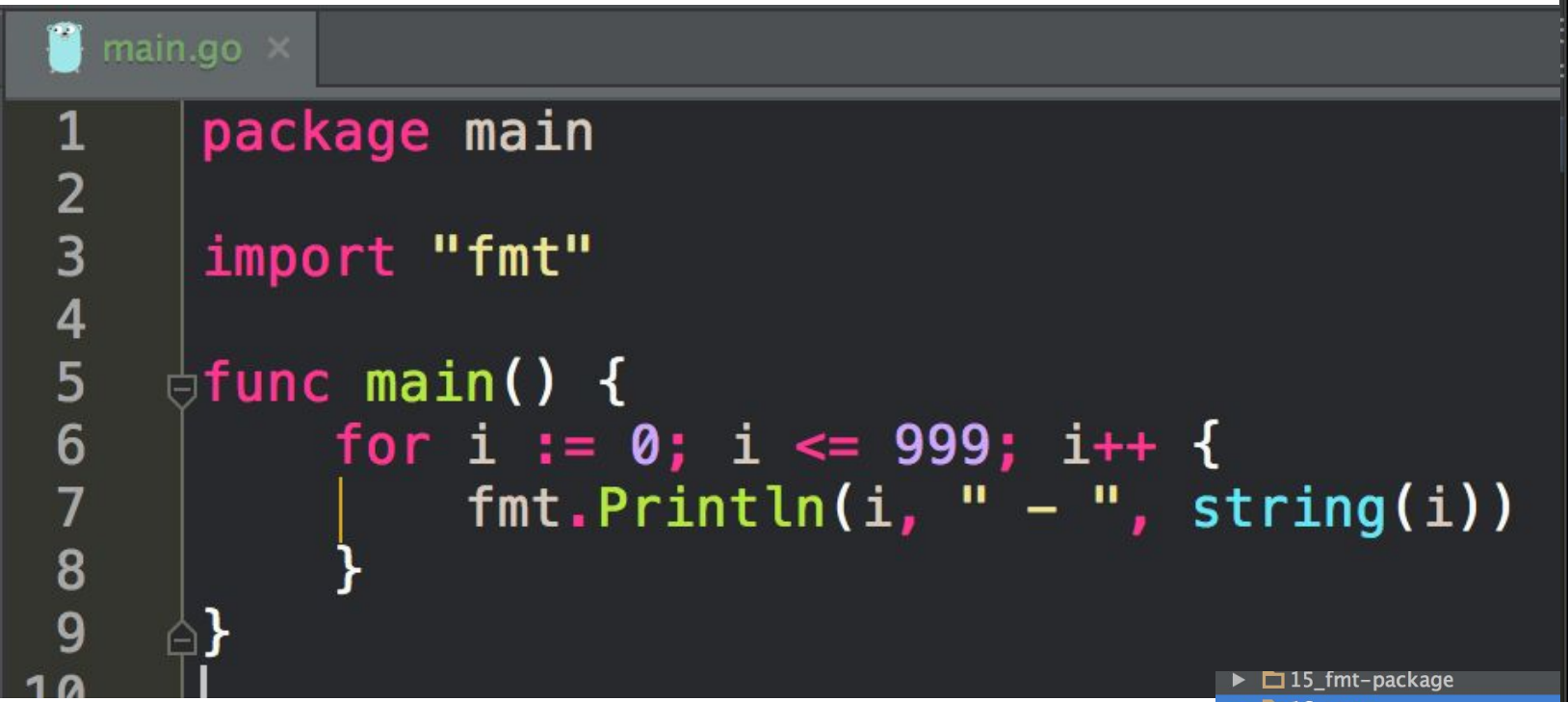
rune is an alias for int32
because UTF-8 is 4 bytes

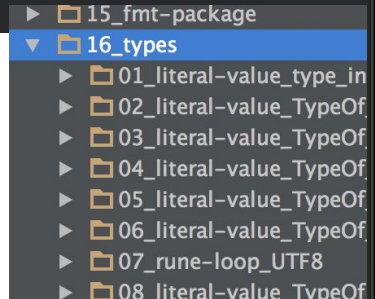
main.go

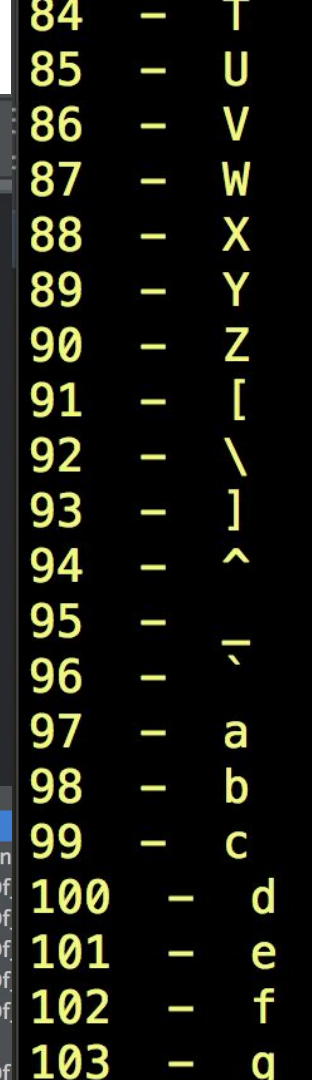
```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    rune := 'd'
    fmt.Printf("%T\n", rune)
    fmt.Println(reflect.TypeOf(rune)," - ",rune)
}
```

Terminal

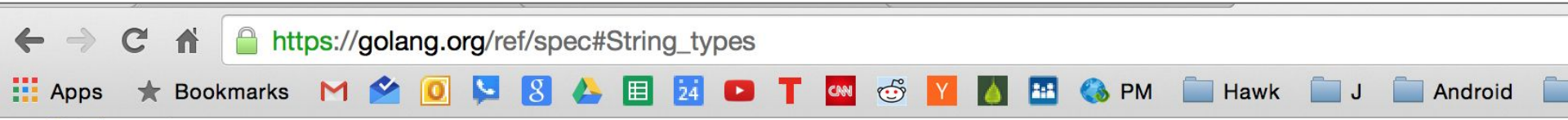
```
05_literal-value_TypeOf_rune_int32 $ go run main.go
int32
int32  -  100
05_literal-value_TypeOf_rune_int32 $
```


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 for i := 0; i <= 999; i++ {
7 fmt.Println(i, " - ", string(i))
8 }
9 }
10


15_fmt-package
16_types
01_literal-value_type_in
02_literal-value_TypeOf
03_literal-value_TypeOf
04_literal-value_TypeOf
05_literal-value_TypeOf
06_literal-value_TypeOf
07_rune-loop_UTF8
08_literal-value_TypeOf


84 - T
85 - U
86 - V
87 - W
88 - X
89 - Y
90 - Z
91 - [
92 - \
93 -]
94 - ^
95 - _
96 - `
97 - a
98 - b
99 - c
100 - d
101 - e
102 - f
103 - g

string

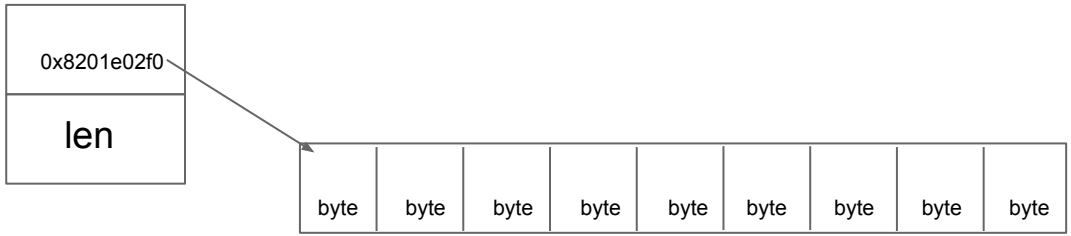


String types

A *string type* represents the set of string values. A string value is a (possibly empty) sequence of bytes. Strings are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is `string`.

The length of a string `s` (its size in bytes) can be discovered using the built-in function `len`. The length is a compile-time constant if the string is a constant. A string's bytes can be accessed by integer indices 0 through `len(s)-1`. It is illegal to take the address of such an element; if `s[i]` is the `i`'th byte of a string, `&s[i]` is invalid.

string

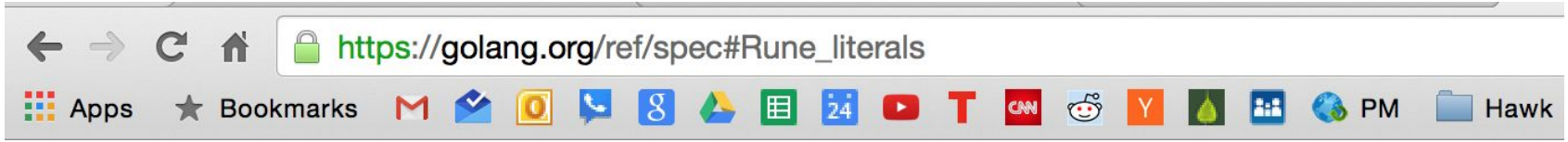


- built-in type
- also a **reference type**
  - reference types are very important
- string is a **two word data structure**
  - either 8 or 16 bytes
    - 32 bit architecture: 4 bytes (32 bits) per word * 2 words = 8 bytes
    - 64 bit architecture: 8 bytes (64 bits) per word * 2 words = 16 bytes
  - first word
    - pointer to array of bytes
  - second word
    - len of those bytes
- all reference types have a pointer
- when we copy a string
  - we are making a copy of the two words:
    - word 1: the pointer to the underlying array of bytes
    - word 2: the length
- strings are immutable
- zero value for a string
  - word 1: nil
    - a pointer set to the value of zero is nil
  - word 2: 0

string

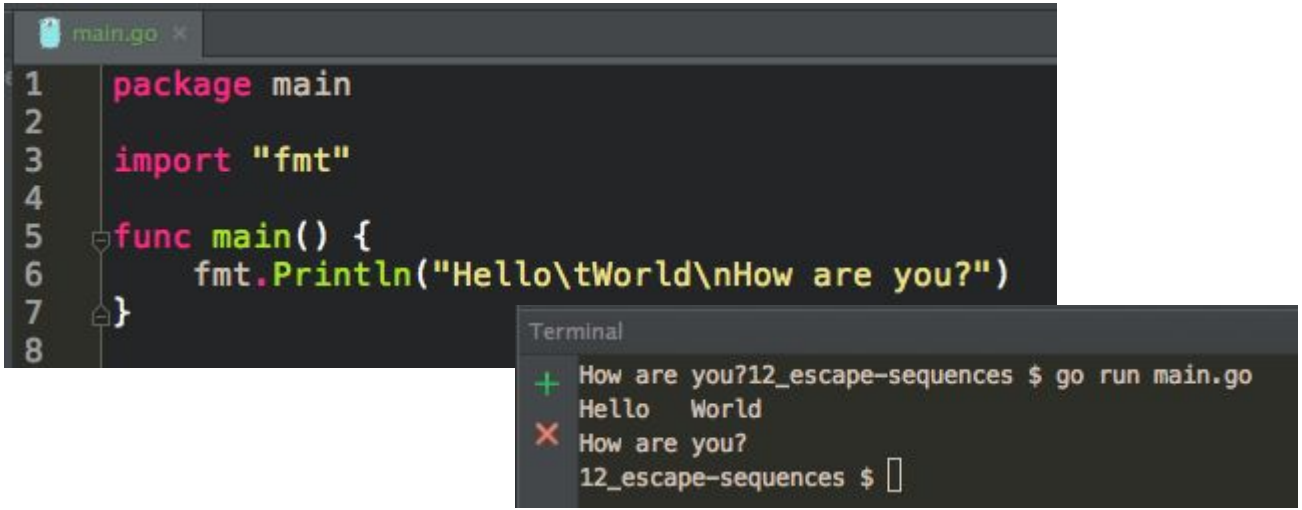
- string
  - series of characters
  - UTF-8 encoding
    - ASCII is a 7 bit character series
      - 128 letters
    - UTF-8 are those same 7 bits, multiple bytes for other characters
      - UTF-8 is a multiple byte character encoding
      - other characters could be two bytes, or three bytes, or four bytes per character
      - chinese characters will probably be four bytes per character;
      - golang uses UTF-8 because Rob Pike & Ken Thompson were partially inventors of UTF-8
  - create a string:
    - double quotes || backticks
    - backticks can be multiple lines
  - escape sequences
    - strings support escape sequences like \n for newline or \t for a tab

escape sequences



After a backslash, certain single-character escapes represent special values:

```
\a   U+0007 alert or bell
\b   U+0008 backspace
\f   U+000C form feed
\n   U+000A line feed or newline
\r   U+000D carriage return
\t   U+0009 horizontal tab
\v   U+000b vertical tab
\\   U+005c backslash
\'   U+0027 single quote  (valid escape only within rune literals)
\"   U+0022 double quote  (valid escape only within string literals)
```


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 fmt.Println("Hello\tWorld\nHow are you?")
7 }
8
Terminal
How are you?12_escape-sequences $ go run main.go
Hello World
How are you?
12_escape-sequences $

main.go

```
1  package main
2  import "fmt"
3
4  func main() {
5      intro := "Four score and seven years ago...."
6      fmt.Println(intro)
7      fmt.Println([]byte(intro)) // sequence of bytes
8  }
```

conversion

Terminal

```
01_sequence-of-bytes $ go run main.go
Four score and seven years ago....
[70 111 117 114 32 115 99 111 114 101 32 97 110 100 32 115 101 118 101 110 32 121 101 97 114 115 32 97 103 111 46 46 46 46]
01_sequence-of-bytes $
```

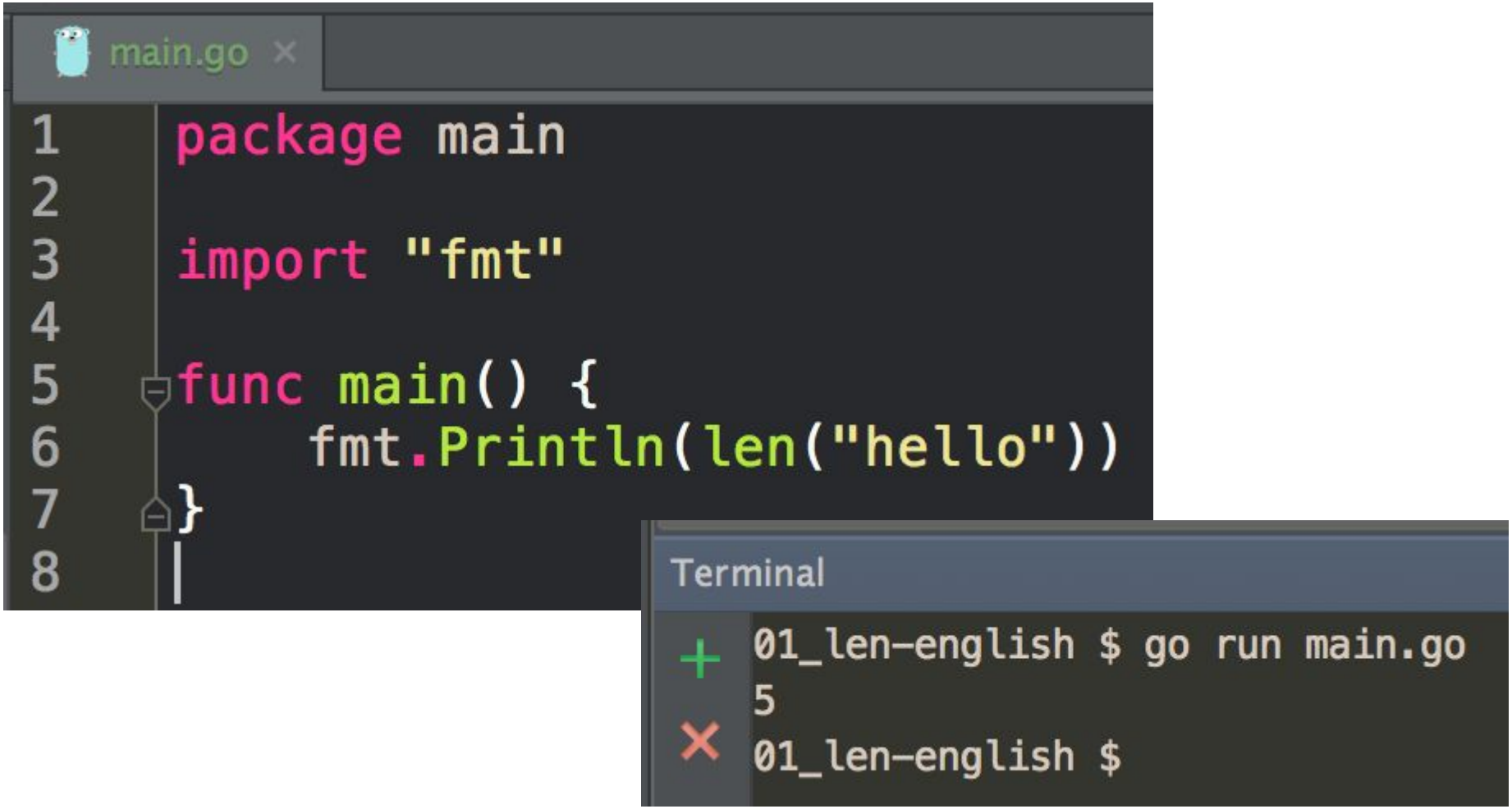
UTF-8
ASCII

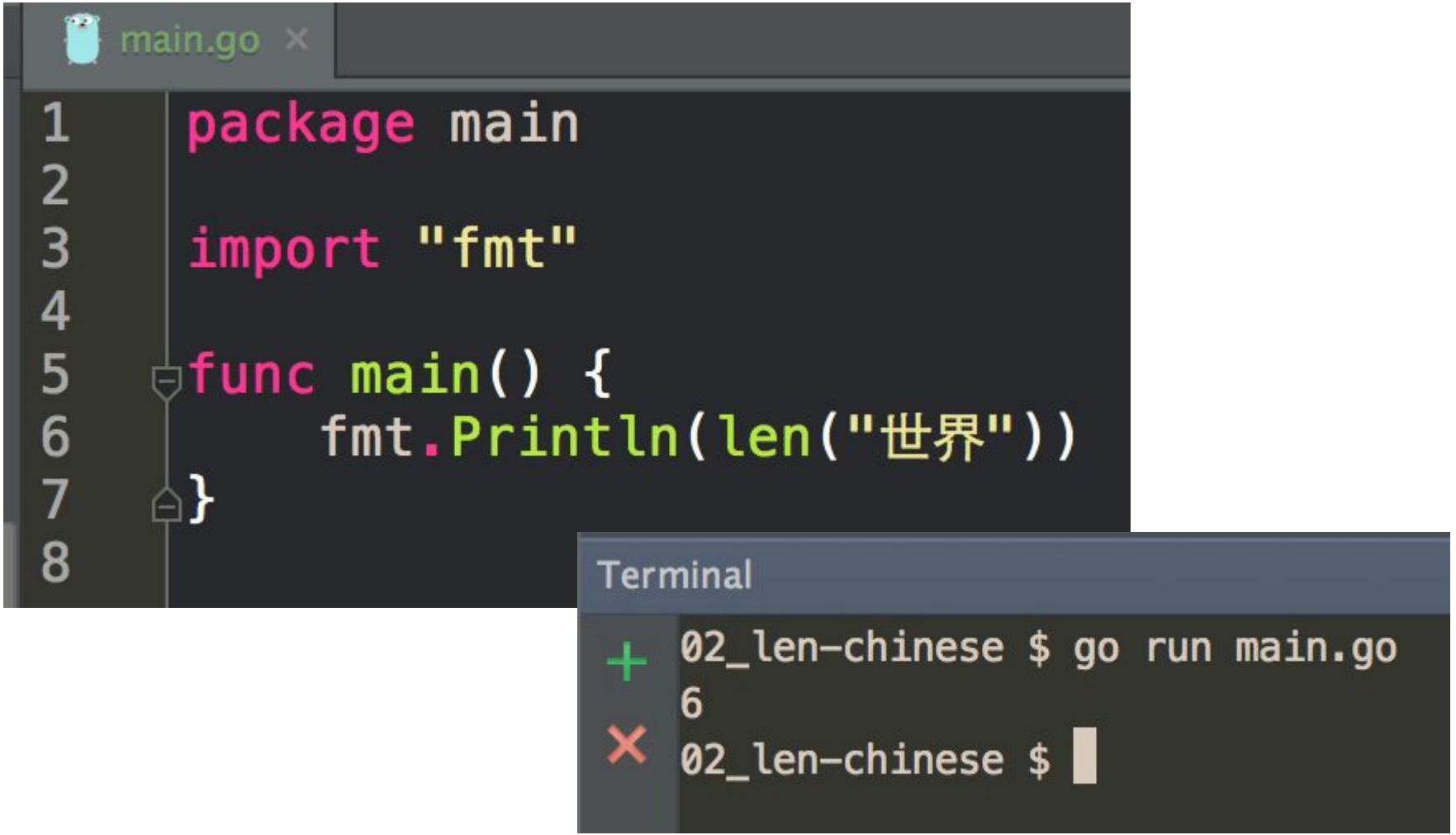
65	41	101	A	A		97	61	141	a	a
66	42	102	B	B		98	62	142	b	b
67	43	103	C	C		99	63	143	c	c
68	44	104	D	D		100	64	144	d	d
69	45	105	E	E		101	65	145	e	e
70	46	106	F	F		102	66	146	f	f
71	47	107	G	G		103	67	147	g	g
72	48	110	H	H		104	68	150	h	h
73	49	111	I	I		105	69	151	i	i
74	4A	112	J	J		106	6A	152	j	j
75	4B	113	K	K		107	6B	153	k	k
76	4C	114	L	L		108	6C	154	l	l
77	4D	115	M	M		109	6D	155	m	m
78	4E	116	N	N		110	6E	156	n	n
79	4F	117	O	O		111	6F	157	o	o
80	50	120	P	P		112	70	160	p	p
81	51	121	Q	Q		113	71	161	q	q
82	52	122	R	R		114	72	162	r	r
83	53	123	S	S		115	73	163	s	s
84	54	124	T	T		116	74	164	t	t
85	55	125	U	U		117	75	165	u	u
86	56	126	V	V		118	76	166	v	v
87	57	127	W	W		119	77	167	w	w
88	58	130	X	X		120	78	170	x	x
89	59	131	Y	Y		121	79	171	y	y
90	5A	132	Z	Z		122	7A	172	z	z
91	5B	133	[	[		123	7B	173	{	{
92	5C	134	\	\		124	7C	174			|
93	5D	135	]	]		125	7D	175	}	}
94	5E	136	^	^		126	7E	176	~	~
95	5F	137	_	_		127	7F	177	&#127;	DEL

string

len

len gives us the number of bytes, not the number of characters


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 fmt.Println(len("hello"))
7 }
8
Terminal
01_len-english $ go run main.go
5
01_len-english $


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 fmt.Println(len("世界"))
7 }
8
Terminal
02_len-chinese $ go run main.go
6
02_len-chinese $

string

index

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      greeting := "Hello"
7      fmt.Println(greeting)
8      fmt.Println(greeting[0])
9      fmt.Println(greeting[4])
10 }
11
```

index

Terminal

```
04_index-access $ go run main.go
Hello
72
111
04_index-access $
```

UTF-8
ASCII

65			5;	A		97	61	141	a	a
66			6;	B		98	62	142	b	b
67			7;	C		99	63	143	c	c
68			8;	D		100	64	144	d	d
69	45	105	E	E		101	65	145	e	e
70	46	106	F	F		102	66	146	f	f
71	47	107	G	G		103	67	147	g	g
72	48	110	H	H		104	68	150	h	h
73	49	111	I	I		105	69	151	i	i
74	4A	112	J	J		106	6A	152	j	j
75	4B	113	K	K		107	6B	153	k	k
76	4C	114	L	L		108	6C	154	l	l
77	4D	115	M	M		109	6D	155	m	m
78	4E	116	N	N		110	6E	156	n	n
79	4F	117	O	O		111	6F	157	o	o
80	50	120	P	P		112	70	160	p	p
81	51	121	Q	Q		113	71	161	q	q
82	52	122	R	R		114	72	162	r	r
83	53	123	S	S		115	73	163	s	s
84	54	124	T	T		116	74	164	t	t
85	55	125	U	U		117	75	165	u	u
86	56	126	V	V		118	76	166	v	v
87	57	127	W	W		119	77	167	w	w
88	58	130	X	X		120	78	170	x	x
89	59	131	Y	Y		121	79	171	y	y
90	5A	132	Z	Z		122	7A	172	z	z
91	5B	133	[	[		123	7B	173	{	{
92	5C	134	\	\		124	7C	174			|
93	5D	135	]	]		125	7D	175	}	}
94	5E	136	^	^		126	7E	176	~	~
95	5F	137	_	_		127	7F	177	&#127;	DEL

string

slicing

main.go

```go
package main

import "fmt"

func main() {
    greeting := "Hello"
    fmt.Println(greeting)
    fmt.Println(greeting[0])
    fmt.Println(greeting[4])
    fmt.Println("-------------")
    fmt.Println("What the ... ")
    fmt.Println(greeting[:4])
    fmt.Println("... did that just do?")
}

```

What is this going to print?

main.go

```go
package main

import "fmt"

func main() {
    greeting := "Hello"
    fmt.Println(greeting)
    fmt.Println(greeting[0])
    fmt.Println(greeting[4])
    fmt.Println("-------------")
    fmt.Println("What the ... ")
    fmt.Println(greeting[:4])
    fmt.Println("... did that just do?")
}
```

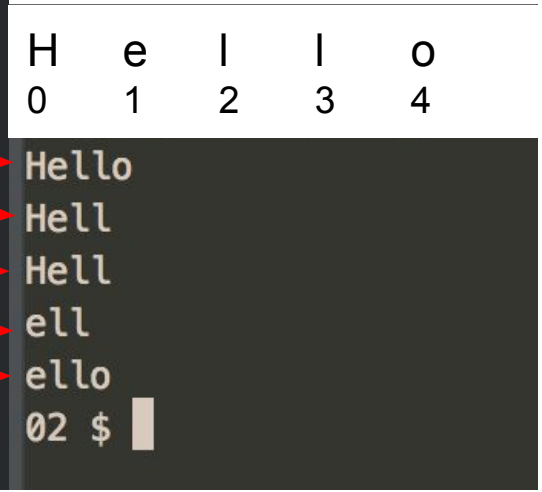
Terminal

```
05_slicing $ go run main.go
Hello
72
111

What the ...
Hell
... did that just do?
05_slicing $
```

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      greeting := "Hello"
7      fmt.Println(greeting)
8      fmt.Println(greeting[:4])
9      fmt.Println(greeting[0:4])
10     fmt.Println(greeting[1:4])
11     fmt.Println(greeting[1:])
12 }
```

slicing

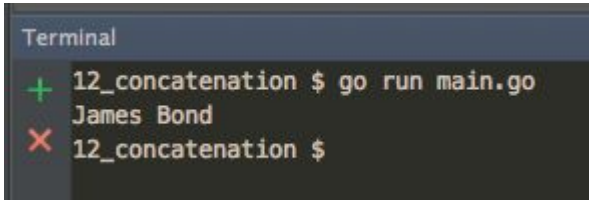


string

concatenation

main.go

```go
package main

import "fmt"

func main() {
    fname := "James"
    lname := "Bond"
    fmt.Println(fname + " " + lname)
}
```



exercise

create a program that uses these **escape sequences**:
\t \n

exercise

create a program that uses **len**

exercise

create a program that using an **index** to access a **rune** in a **string**

exercise

create a program that **slices** a **string**

exercise

create a program that **concatenates** two variables both of type **string**

strconv package

The most common numeric conversions are **Atoi** (string to int) and **Itoa** (int to string).

package strconv

```
import "strconv"
```

Package strconv implements conversions to and from string representations of basic data types.

Numeric Conversions

The most common numeric conversions are Atoi (string to int) and Itoa (int to string).

```
i, err := strconv.Atoi("-42")
s := strconv.Itoa(-42)
```

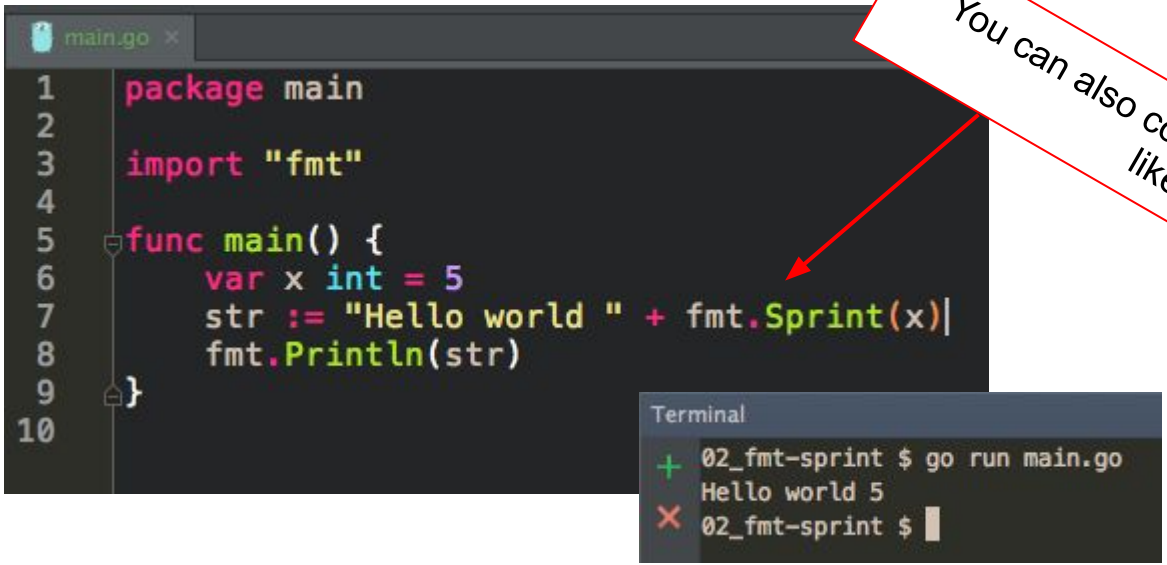
These assume decimal and the Go int type

main.go

```go
package main
import (
    "strconv"
    "fmt"
)

func main() {
    var x int = 5
    str := "Hello world " + strconv.Itoa(x) // int to ascii
    fmt.Println(str)
}

```

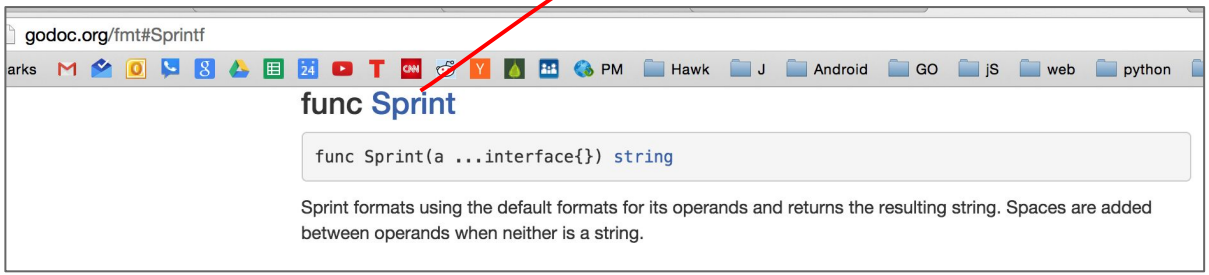
Terminal

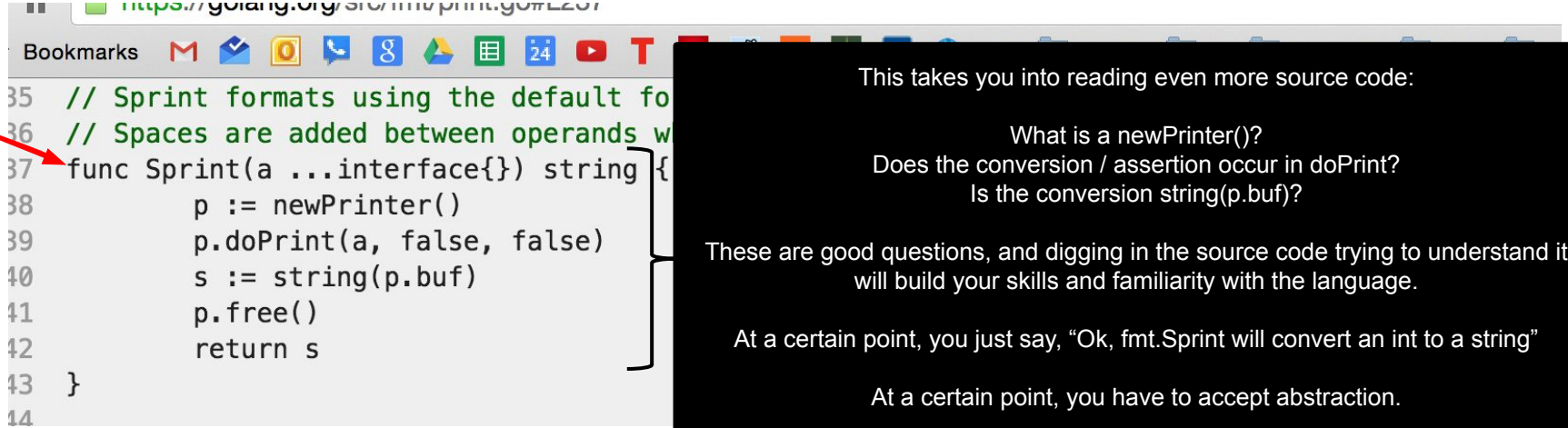
```
01_itoa $ go run main.go
Hello world 5
01_itoa $
```



Let's look up fmt.Sprint to see how it converts an int to a string

click the name of the func and you are taken to the source code





This takes you into reading even more source code:

What is a newPrinter()?
Does the conversion / assertion occur in doPrint?
Is the conversion string(p.buf)?

These are good questions, and digging in the source code trying to understand it will build your skills and familiarity with the language.

At a certain point, you just say, “Ok, fmt.Sprint will convert an int to a string”

At a certain point, you have to accept abstraction.

“If you wish to make apple pie from scratch, you must first create the universe”
~ Carl Sagan

main.go

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    i, _ := strconv.Atoi("32")
    fmt.Println(i+10)
}

```

Terminal

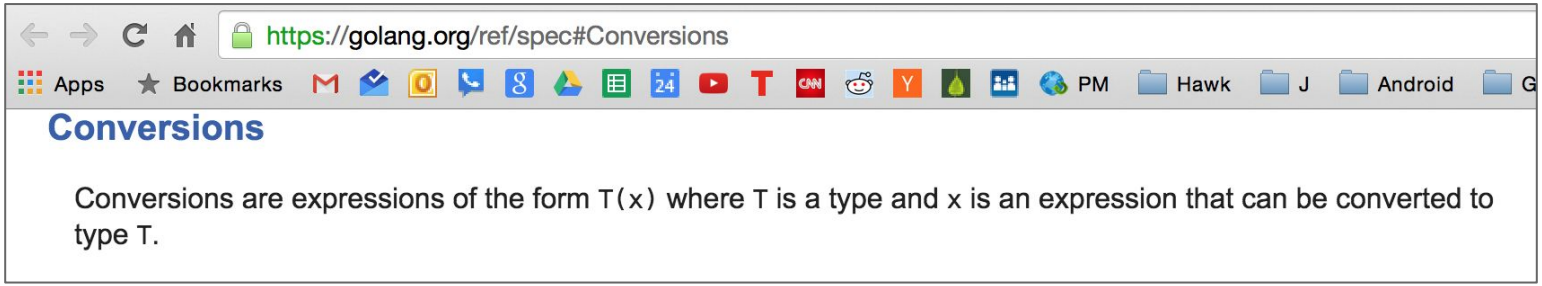
```
03_atoi $ go run main.go
42
03_atoi $
```

exercise

create a program that uses **Atoi** & **Itoa**

conversion

known as casting in some languages


https://golang.org/ref/spec#Conversions
Apps
Bookmarks
PM
Hawk
J
Android
G
Conversions
Conversions are expressions of the form T(x) where T is a type and x is an expression that can be converted to type T.

for you, dear student, to read on your own ....

Conversions between numeric types

For the conversion of non-constant numeric values, the following rules apply:

1. When converting between integer types, if the value is a signed integer, it is sign extended to implicit infinite precision; is then truncated to fit in the result type's size. For example, if `v := uint16(0x10F0)`, then `uint32(int8(v)) == 0xFFFFFFF0`. The conversion always yields tion of overflow.
2. When converting a floating-point number to an integer, the fraction is discarded (truncation towards zero).
3. When converting an integer or floating-point number to a floating-point type, or a complex number to another complex type, the ecision specified by the destination type. For instance, the value of a variable x of type `float32` may be stored using additional precision beyond that of an at32(x) represents the result of rounding x's value to 32-bit precision. Similarly, `x + 0.1` may use more than 32 bits of precision, but `float32(x + 0.1)`

In all non-constant conversions involving floating-point or complex values, if the result type cannot represent the value the conversion succeeds but the result value is implementation-dependent.

Conversions to and from a string type

1. Converting a signed or unsigned integer value to a string type yields a string containing the UTF-8 representation of the integer. Values outside the range are converted to `"\uFFFD"`.

```
string('a')       // "a"
string(-1)        // "\ufffd" == "\xef\xbf\xbd"
string(0xf8)      // "\u00f8" == "ø" == "\xc3\xb8"
type MyString string
MyString(0x65e5)  // "\u65e5" == "日" == "\xe6\x97\xa5"
```

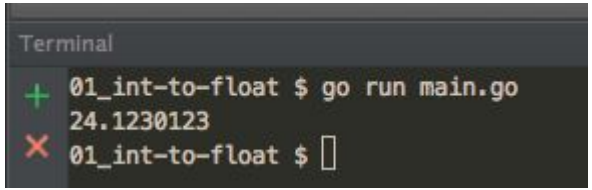
2. Converting a slice of bytes to a string type yields a string whose successive bytes are the elements of the slice.

```
string([]byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'})   // "hellø"
string([]byte{})                                     // ""
string([]byte(nil))                                  // ""

type MyBytes []byte
string(MyBytes{'h', 'e', 'l', 'l', '\xc3', '\xb8'})  // "hellø"
```

3. Converting a slice of runes to a string type yields a string that is the concatenation of the individual rune values converted to strings.

main.go

```go
package main
import "fmt"

func main(){
    var x int = 12
    var y float64 = 12.1230123
    fmt.Println(y + float64(x))
    // conversion: int to float64
}
```

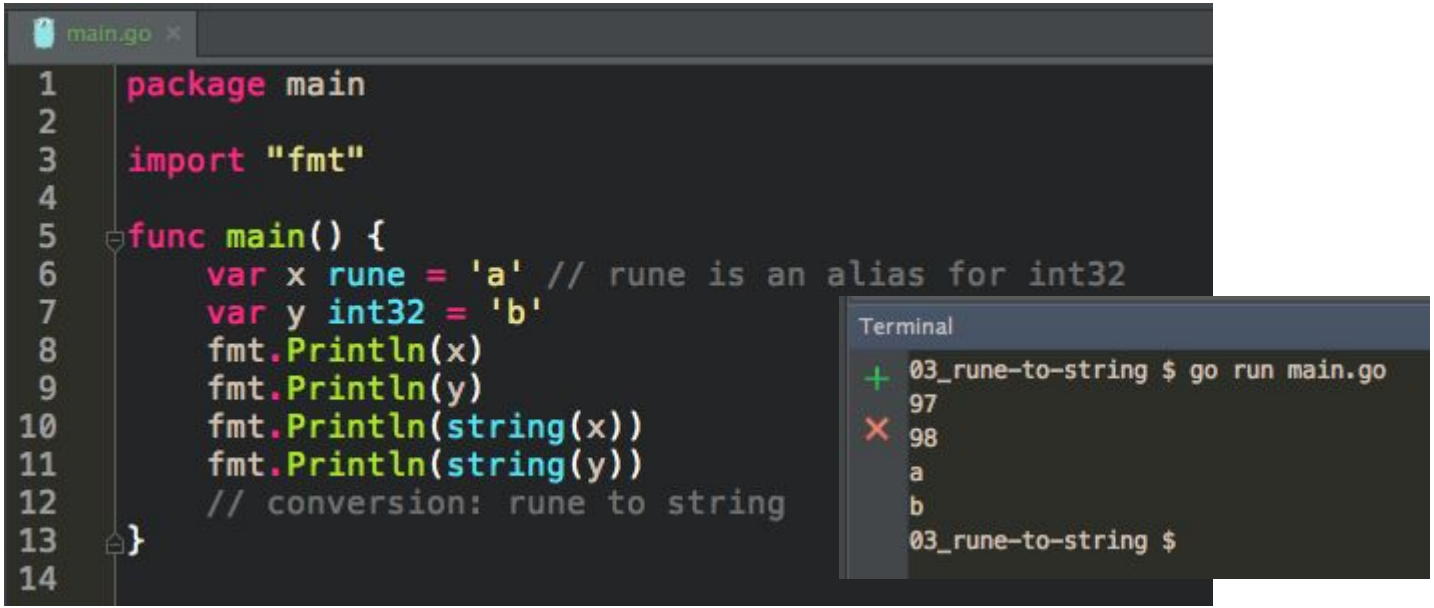


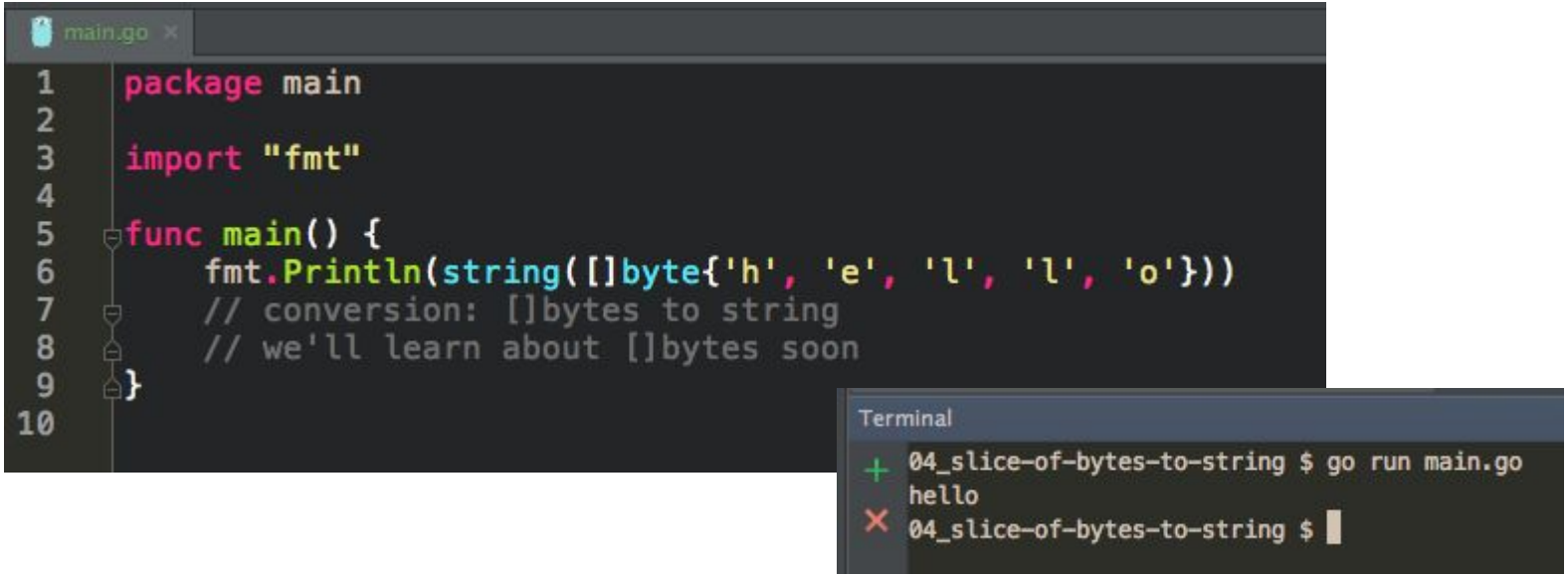
main.go

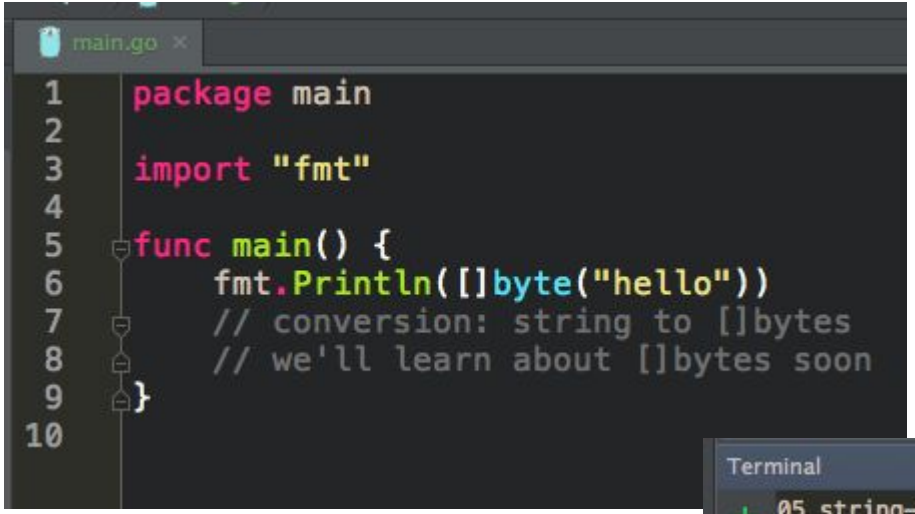
```go
package main

import "fmt"

func main() {
    var x int = 12
    var y float64 = 12.1230123
    fmt.Println(int(y) + x)
    // conversion: float64 to int
}

```

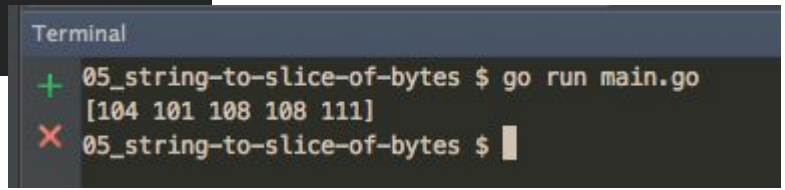
Terminal

```
02_float-to-int $ go run main.go
24
02_float-to-int $
```


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 var x rune = 'a' // rune is an alias for int32
7 var y int32 = 'b'
8 fmt.Println(x)
9 fmt.Println(y)
10 fmt.Println(string(x))
11 fmt.Println(string(y))
12 // conversion: rune to string
13 }
14
Terminal
03_rune-to-string $ go run main.go
97
98
a
b
03_rune-to-string $


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 fmt.Println(string([]byte{'h', 'e', 'l', 'l', 'o'}))
7 // conversion: []bytes to string
8 // we'll learn about []bytes soon
9 }
10
Terminal
04_slice-of-bytes-to-string $ go run main.go
hello
04_slice-of-bytes-to-string $


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 fmt.Println([]byte("hello"))
7 // conversion: string to []bytes
8 // we'll learn about []bytes soon
9 }
10


Terminal
05_string-to-slice-of-bytes $ go run main.go
[104 101 108 108 111]
05_string-to-slice-of-bytes $

exercise

create a program that converts int to float64

exercise

create a program that converts float64 to int

exercise

create a program that converts []byte to string

exercise

create a program that converts string to []byte

assertion

conversion - change one type to another
assertion - used for interfaces

assertion

conversion - change one type to another
assertion - used for interfaces

We will learn more about interfaces.
As you gain exposure to them, you will gain understanding.

type.go

```go
package main

import "fmt"

//  switch on types
//  -- normally we switch on value of variable
//  -- go allows you to switch on type of variable

type Contact struct {
    greeting string
    name     string
}

// we'll learn more about interfaces later
func SwitchOnType(x interface{}) {
    switch x.(type) {       // this is an assert; asserting, "x is of this type"
    case int:
        fmt.Println("int")
    case string:
        fmt.Println("string")
    case Contact:
        fmt.Println("contact")
    default:
        fmt.Println("unknown")

    }
}

func main() {
    SwitchOnType(7)
    SwitchOnType("McLeod")
    var t = Contact{"Good to see you,", "Tim"}
    SwitchOnType(t)
    SwitchOnType(t.greeting)
    SwitchOnType(t.name)
}
```

notice x is of type interface{}

Already Saw This

Terminal

```
05_on-type $ go run type.g
int
string
contact
string
string
05_on-type $
```

Type assertions

For an expression x of interface type and a type T, the primary expression

```
x.(T)
```

asserts that x is not nil and that the value stored in x is of type T. The notation x.(T) is called a *type assertion*.

More precisely, if T is not an interface type, x.(T) asserts that the dynamic type of x is identical to the type T. In this case, T must implement the (interface) type of x; otherwise the type assertion is invalid since it is not possible for x to store a value of type T. If T is an interface type, x.(T) asserts that the dynamic type of x implements the interface T.

assertions
are for interface types

If the type assertion holds, the value of the expression is the value stored in x and its type is T. If the type assertion is false, a run-time panic occurs. In other words, even though the dynamic type of x is known only at run time, the type of x.(T) is known to be T in a correct program.

```
var x interface{} = 7  // x has dynamic type int and value 7
i := x.(int)           // i has type int and value 7

type I interface { m() }
var y I
s := y.(string)        // illegal: string does not implement I (missing method m)
r := y.(io.Reader)     // r has type io.Reader and y must implement both I and io.Reader
```

dynamic type
even though go is statically typed
we can use interfaces to create dynamic typing

A type assertion used in an assignment or initialization of the special form

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
```

yields an additional untyped boolean value. The value of ok is true if the assertion holds. Otherwise it is false and the value of v is the zero value for type T. No run-time panic occurs in this case.


```
str, ok := value.(string)
if ok {
    fmt.Printf("string value is: %q\n", str)
} else {
    fmt.Printf("value is not a string\n")
}
```

If the type assertion fails, `str` will still exist and be of type string, but it will have the zero value, an empty string.

main.go

```go
package main

import "fmt"

func main() {
    name := "Sydney"
    str, ok := name.(string)
    if ok {
        fmt.Printf("%q\n", str)
    } else {
        fmt.Printf("value is not a string\n")
    }
}
```

name is not of type interface{}

assertions
are for interface types

Terminal

```
13_assertion $ go run main.go
# command-line-arguments
./main.go:7: invalid type assertion: fname.(string) (non-interface type string on left)
13_assertion $
```

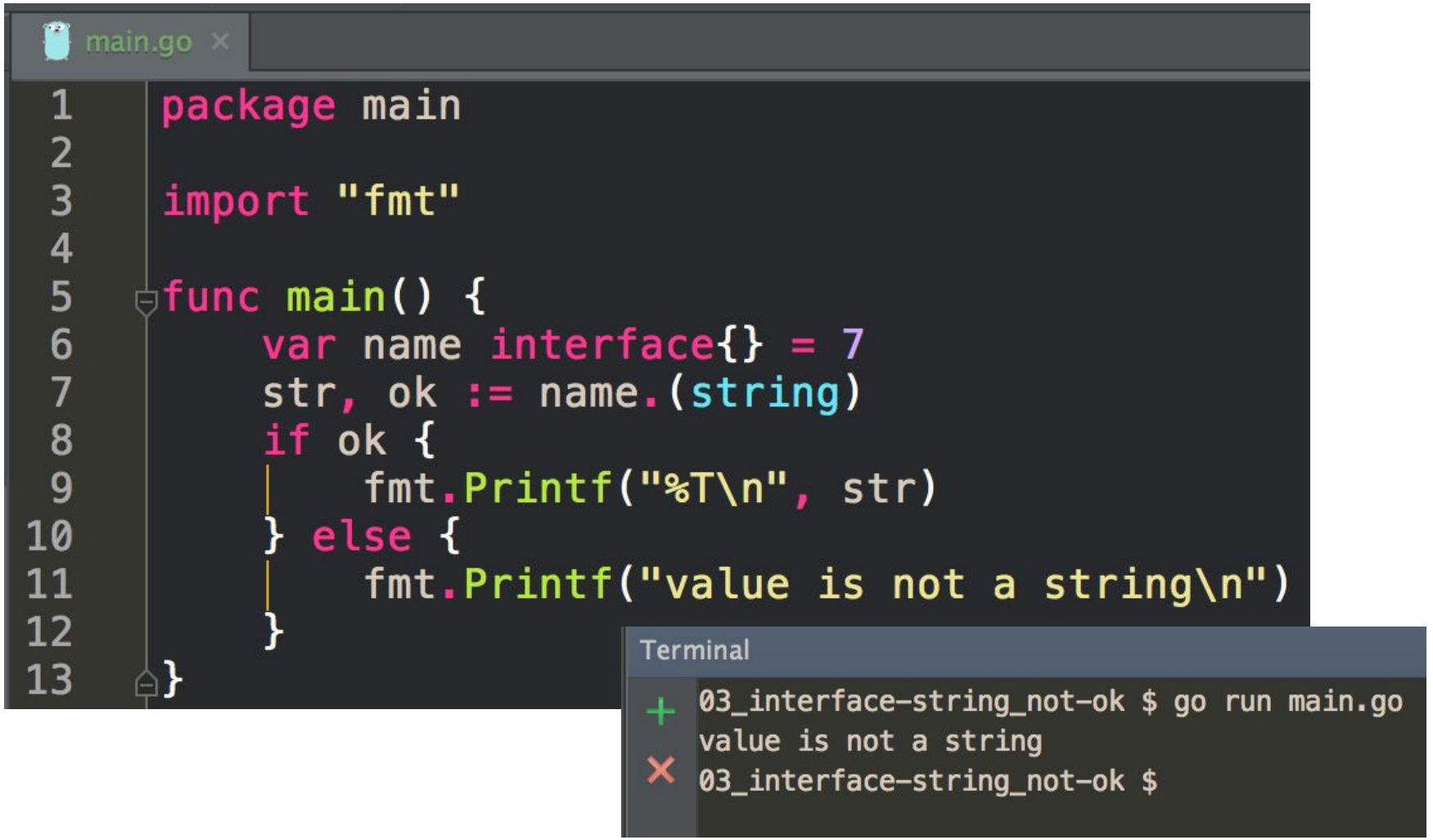
main.go

```go
package main

import "fmt"

func main() {
    var name interface{} = "Sydney"
    str, ok := name.(string)
    if ok {
        fmt.Printf("%T\n", str)
    } else {
        fmt.Printf("value is not a string\n")
    }
}
```

assertions
are for interface types

Terminal

```
02_interface-string $ go run main.go
string
02_interface-string $
```


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 var name interface{} = 7
7 str, ok := name.(string)
8 if ok {
9 fmt.Printf("%T\n", str)
10 } else {
11 fmt.Printf("value is not a string\n")
12 }
13 }
Terminal
03_interface-string_not-ok $ go run main.go
value is not a string
03_interface-string_not-ok $

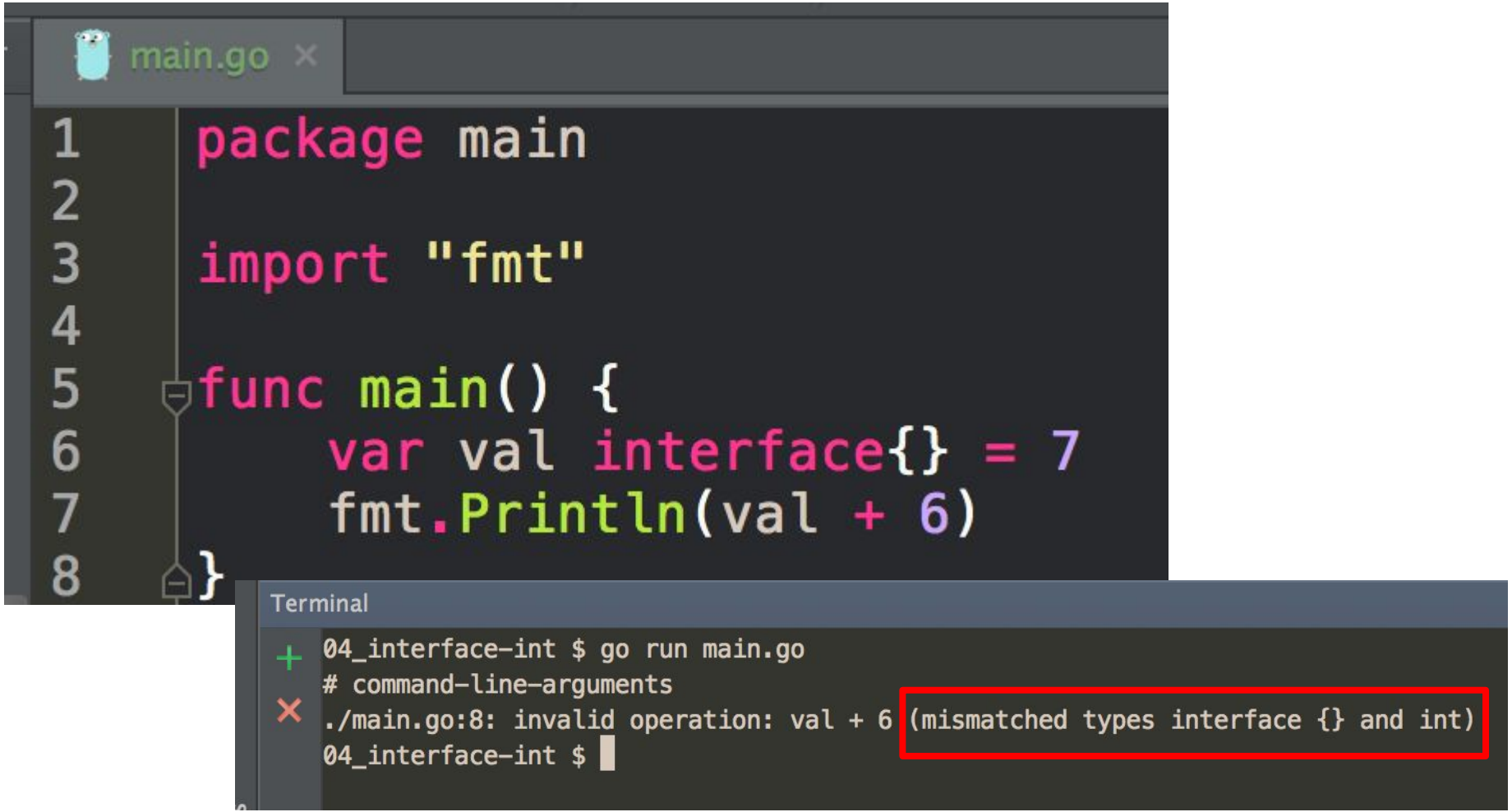
main.go

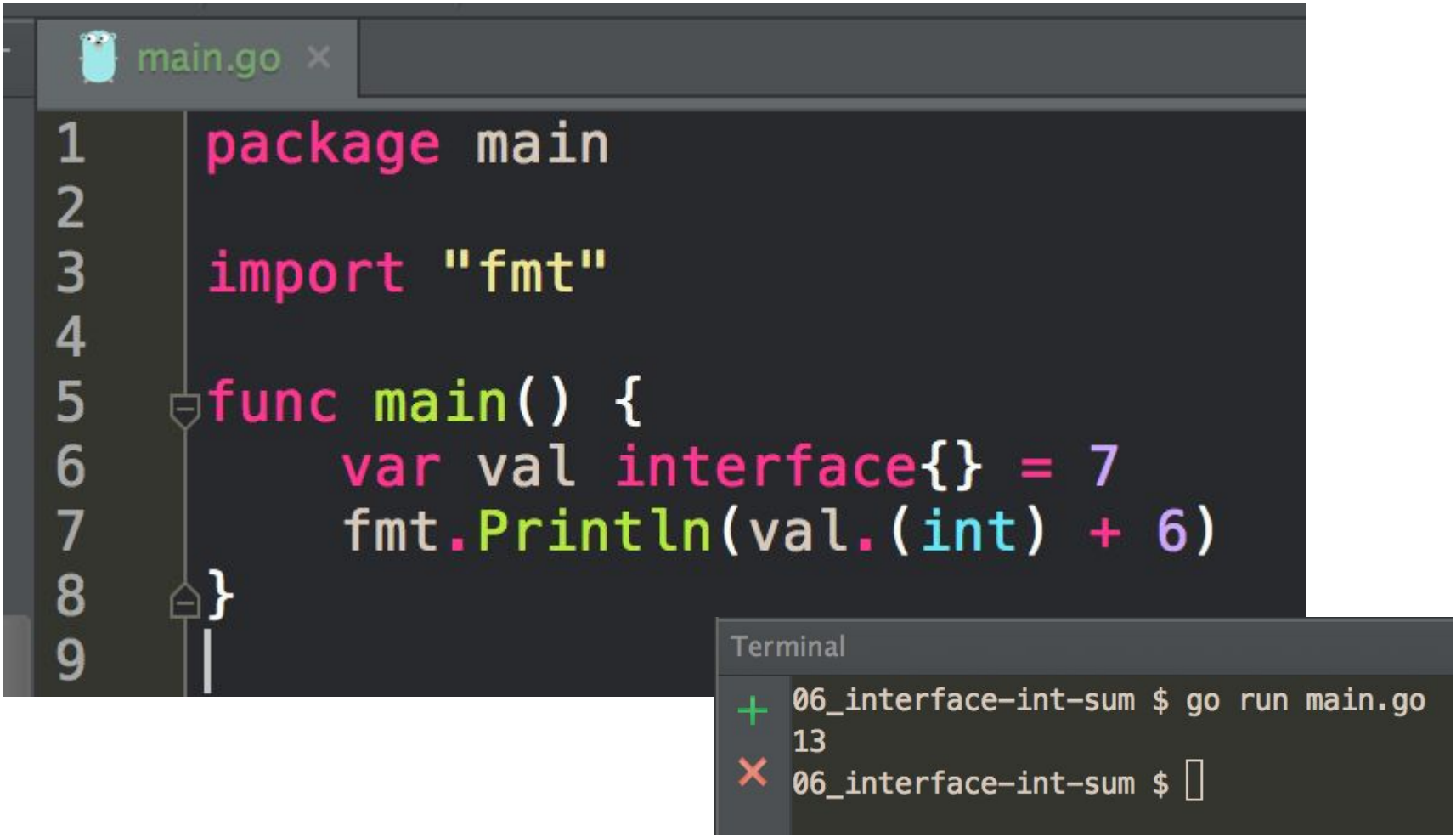
```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var val interface{} = 7
7      fmt.Printf("%T\n", val)
8  //  fmt.Println(val + 6)
9  }
```

What will happen when I uncomment line 8 and run this?

Terminal

```
04_interface-int $ go run main.go
int
04_interface-int $
```


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 var val interface{} = 7
7 fmt.Println(val + 6)
8 }
Terminal
04_interface-int $ go run main.go
command-line-arguments
./main.go:8: invalid operation: val + 6 (mismatched types interface {} and int)
04_interface-int $


main.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 var val interface{} = 7
7 fmt.Println(val.(int) + 6)
8 }
9
Terminal
06_interface-int-sum $ go run main.go
13
06_interface-int-sum $

main.go

```go
package main

import "fmt"

func main() {
	rem := 7.24
	fmt.Printf("%T\n", rem)
	fmt.Printf("%T\n", int(rem))
}
```

reminder:
casting

Terminal

```
07_casting-reminder $ go run main.go
float64
int
07_casting-reminder $
```

main.go

```go
package main

import "fmt"

func main() {
	rem := 7.24
	fmt.Printf("%T\n", rem)
	fmt.Printf("%T\n", int(rem))

	var val interface{} = 7
	fmt.Printf("%T\n", val)
	fmt.Printf("%T\n", int(val))
}
```

can we cast
type interface?

Terminal

```
08_interface-cast-error_need-type-assertion $ go run main.go
# command-line-arguments
./main.go:12: cannot convert val (type interface {}) to type int: need type assertion
08_interface-cast-error_need-type-assertion $
```

BACK TO TYPE

Types

- Method sets
- Boolean types
- Numeric types
- String types
- Array types
- Slice types
- Struct types
- Pointer types
- Function types
- Interface types
- Map types
- Channel types

https://golang.org/ref/spec

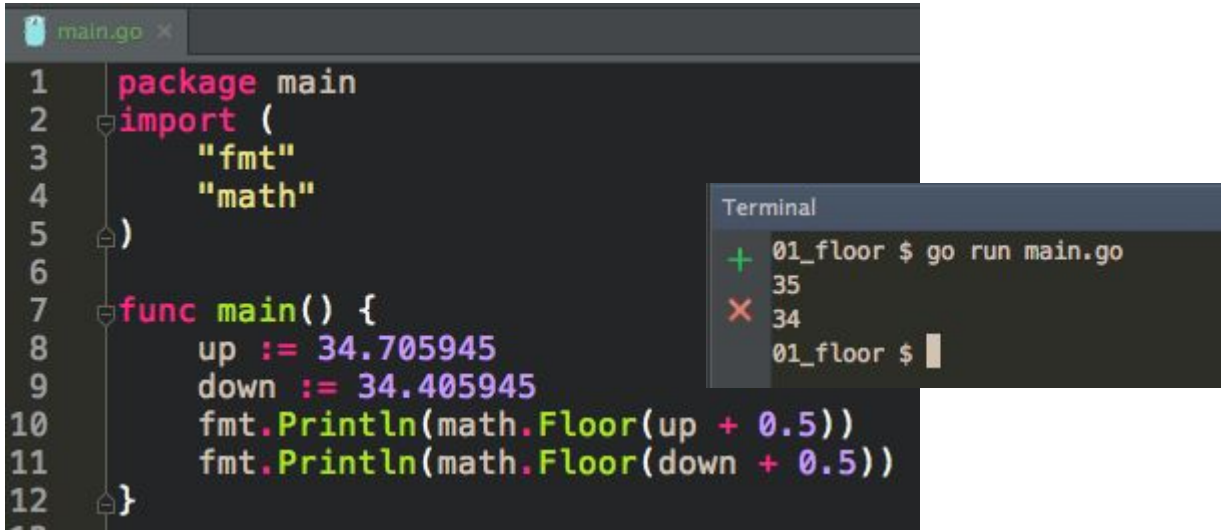
math package

package math

```
import "math"
```

Package math provides basic constants and mathematical functions.

Index


main.go
1 package main
2 import (
3 "fmt"
4 "math"
5)
6
7 func main() {
8 up := 34.705945
9 down := 34.405945
10 fmt.Println(math.Floor(up + 0.5))
11 fmt.Println(math.Floor(down + 0.5))
12 }
Terminal
01_floor $ go run main.go
35
34
01_floor $

exercise

rounding up
create a program that asks the user for a float64 and assigns it to x
then print the least integer value greater than or equal to x
for example, if someone entered 43.2, print: 44
for example, if someone entered 43.9, print: 44
use the math pkg func **Ceil**

reading docs

an exercise ...

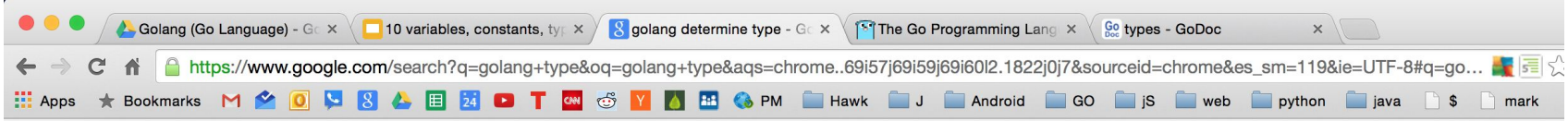
exercise

how would you learn about
TypeOf

```go
package main

import (
    "fmt"
    "reflect"
)

var a string = "this is stored in the variable a" // pa
var b, c string = "stored in b", "stored in c" // packa
var d string // package scope

func main() {

    d = "stored in d" // declaration above; assignment
    var e int = 42 // function scope - subsequent varia
    f := 43
    g := "stored in g"
    h, i := "stored in h", "stored in i"
    j, k, l, m := 44.7, true, false, 'm' // single quot
    n := "n" // double quotes
    o := `o` // back ticks

    fmt.Println("a - ", reflect.TypeOf(a), " - ", a)
    fmt.Println("b - ", reflect.TypeOf(b), " - ", b)
    fmt.Println("c - ", reflect.TypeOf(c), " - ", c)
```

Terminal

```
01_variables $ cd ../02_typeOf/
02_typeOf $ go run typeOf.go
a -  string  -  this is stored in the variable a
b -  string  -  stored in b
c -  string  -  stored in c
d -  string  -  stored in d
e -  int  -  42
f -  int  -  43
g -  string  -  stored in g
h -  string  -  stored in h
i -  string  -  stored in i
j -  float64  -  44.7
k -  bool  -  true
l -  bool  -  false
m -  int32  -  109
n -  string  -  n
o -  string  -  o
02_typeOf $
```



About 87,400 results (0.42 seconds)

go - How to find a type of a object in golang? - Stack Overflow

stackoverflow.com/questions/.../how-to-find-a-**type**-of-a-object-in-**golan**...

Nov 24, 2013 - How Do I find **type** of an object in **Golang** ? In python i just use typeof ... The Go reflection package has methods for inspecting the **type** of ...

How to check variable type at runtime in Go language ...

stackoverflow.com/.../how-to-**check**-variable-**type**-at-runtime-in-**go-lang**...

Aug 9, 2011 - so I need to be able to **check** param **type** at runtime. How do I do that and is ... **type** assertions here: http://**golang**.org/ref/spec#Type_assertions.

reflect - The Go Programming Language

golang.org/pkg/reflect/ Go

The typical use is to take a value with static **type** interface{} and extract its dynamic **type** ... to reflection in Go: http://**golang**.org/doc/articles/laws_of_reflection.html ...

You've visited this page 2 times. Last visit: 8/17/15

The Go Programming Language Specification - The Go ...

https://**golang**.org/ref/spec Go

Nov 11, 2014 - For more information and other documents, see **golang**.org. ..... A constant may be given a **type** explicitly by a constant declaration or conversion, ... At package level, initialization dependencies **determine** the evaluation ...

You've visited this page many times. Last visit: 8/18/15

The Laws of Reflection - The Go Blog

golang.org/blog/laws-of-reflection Go

Sep 6, 2011 - Reflection in computing is the ability of a program to examine its own structure, particularly through **types**; it's a form of metaprogramming.

Web Shopping News Images Videos More Search tools

About 247,000 results (0.38 seconds)

reflect - The Go Programming Language

golang.org/pkg/reflect/ Go

See http://**golang**.org/ref/spec#Uniqueness_of_identifiers Name string PkgPath string Type .... **TypeOf** returns the reflection **Type of** the value in the interface{}.

The Go Programming Language Specification - The Go ...

https://**golang**.org/ref/spec Go

Nov 11, 2014 - For more information and other documents, see **golang**.org. ..... The default **type of** an untyped constant is bool , rune , int , float64 , complex128 ...

You've visited this page many times. Last visit: 8/16/15

The Laws of Reflection - The Go Blog

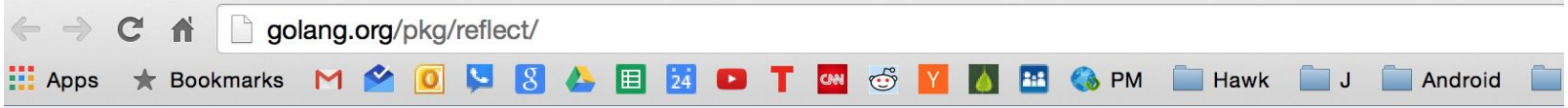
golang.org/blog/laws-of-reflection Go

Sep 6, 2011 - **golang**.org · Install Go · A Tour of Go · Go Documentation · Go ... Reader : Go is statically typed and the static **type of** r is io. ... **TypeOf** returns the reflection **Type of** the value in the interface{}. func **TypeOf**(i interface{}) Type.

go - How to find a type of a object in golang? - Stack Overflow

stackoverflow.com/questions/.../how-to-find-a-**type-of**-a-object-in-**golan**...

Nov 24, 2013 - How Do I find **type of** an object in **Golang** ? In python i just use **typeof**



The Go Programming Language

Documen

Package reflect

```
import "reflect"
```

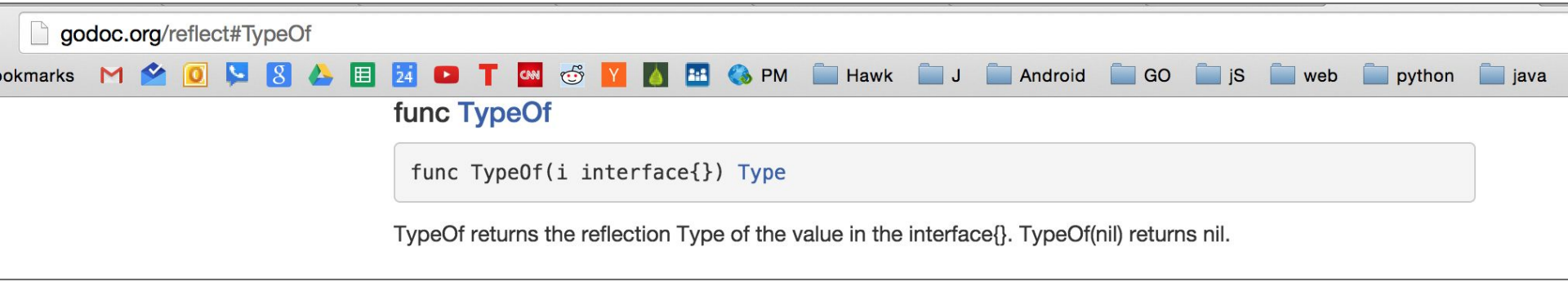
Overview
Index
Examples

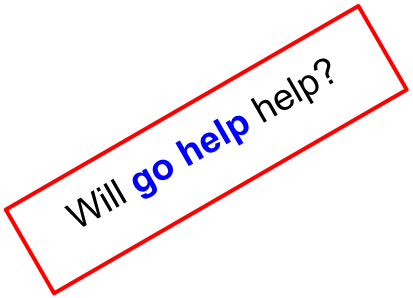
Overview ▾

Package reflect implements run-time reflection, allowing a program to manipulate objects with arbitrary types. The typical use is to take a value with static type interface{} and extract its dynamic type information by calling TypeOf, which returns a Type.

A call to ValueOf returns a Value representing the run-time data. Zero takes a Type and returns a Value representing a zero value for that type.

See "The Laws of Reflection" for an introduction to reflection in Go:
http://golang.org/doc/articles/laws_of_reflection.html


godoc.org/reflect#TypeOf
ookmarks
PM
Hawk
J
Android
GO
jS
web
python
java
func TypeOf
func TypeOf(i interface{}) Type
TypeOf returns the reflection Type of the value in the interface{}. TypeOf(nil) returns nil.



tm002 — bash — 122×38

```
~ $ go help
Go is a tool for managing Go source code.

Usage:

        go command [arguments]

The commands are:

    build       compile packages and dependencies
    clean       remove object files
    env         print Go environment information
    fix         run go tool fix on packages
    fmt         run gofmt on package sources
    generate    generate Go files by processing source
    get         download and install packages and dependencies
    install     compile and install packages and dependencies
    list        list packages
    run         compile and run Go program
    test        test packages
    tool        run specified go tool
    version     print Go version
    vet         run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

    c           calling between Go and C
    filetype    file types
    gopath      GOPATH environment variable
    importpath  import path syntax
    packages    description of package lists
    testflag    description of testing flags
    testfunc    description of testing functions

Use "go help [topic]" for more information about that topic.
```


```
tm002 — bash — 122×38
~ $ go help
Go is a tool for managing Go source code.

Usage:

        go command [arguments]

The commands are:

    build       compile packages and dependencies
    clean       remove object files
    env         print Go environment information
    fix         run go tool fix on packages
    fmt         run gofmt on package sources
    generate    generate Go files by processing source
    get         download and install packages and dependencies
    install     compile and install packages and dependencies
    list        list packages
    run         compile and run Go program
    test        test packages
    tool        run specified go tool
    version     print Go version
    vet         run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

    c           calling between Go and C
    filetype    file types
    gopath      GOPATH environment variable
    importpath  import path syntax
    packages    description of package lists
    testflag    description of testing flags
    testfunc    description of testing functions

Use "go help [topic]" for more information about that topic.
```

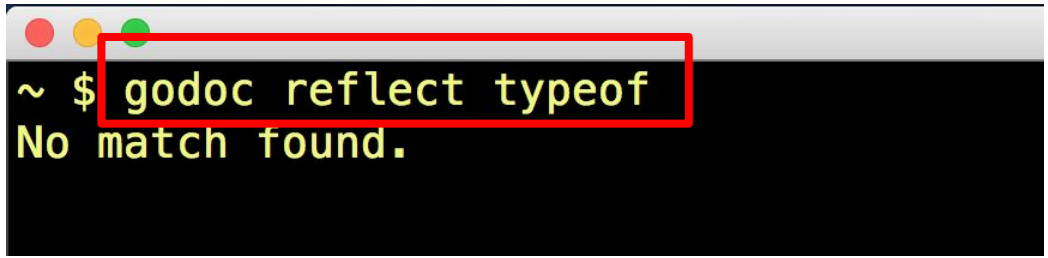
true or false

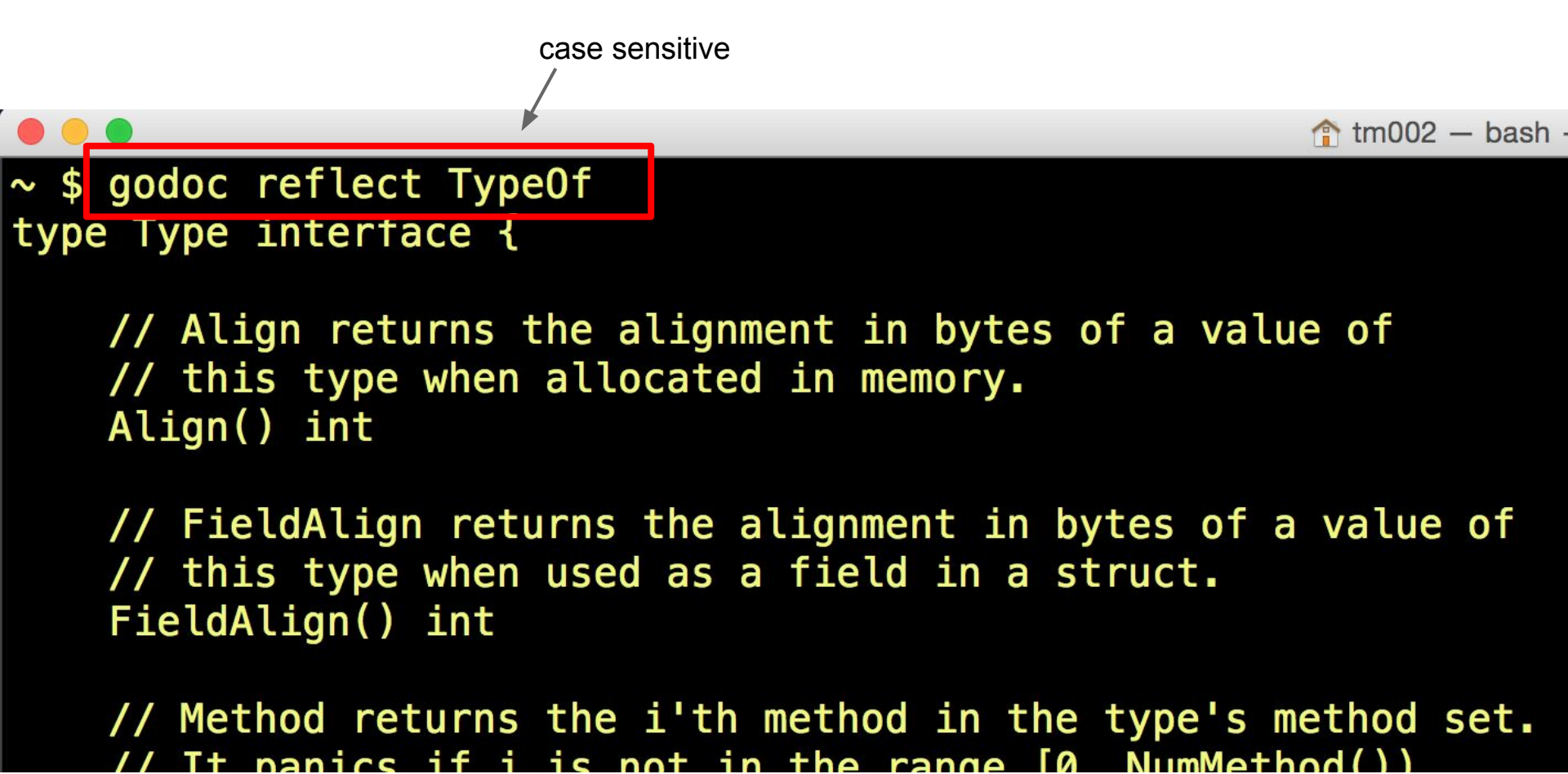
godoc at the terminal can bring up
the documentation for the **reflect** package

```
~ $ godoc reflect
PACKAGE DOCUMENTATION

package reflect
    import "reflect"

    Package reflect implements run-time reflection, allowing a program to
    manipulate objects with arbitrary types. The typical use is to take a
    value with static type interface{} and extract its dynamic type
    information by calling TypeOf, which returns a Type.

    A call to ValueOf returns a Value representing the run-time data. Zero
    takes a Type and returns a Value representing a zero value for that
    type.

    See "The Laws of Reflection" for an introduction to reflection in Go:
    http://golang.org/doc/articles/laws_of_reflection.html

FUNCTIONS

func Copy(dst, src Value) int
    Copy copies the contents of src into dst until either dst has been
```

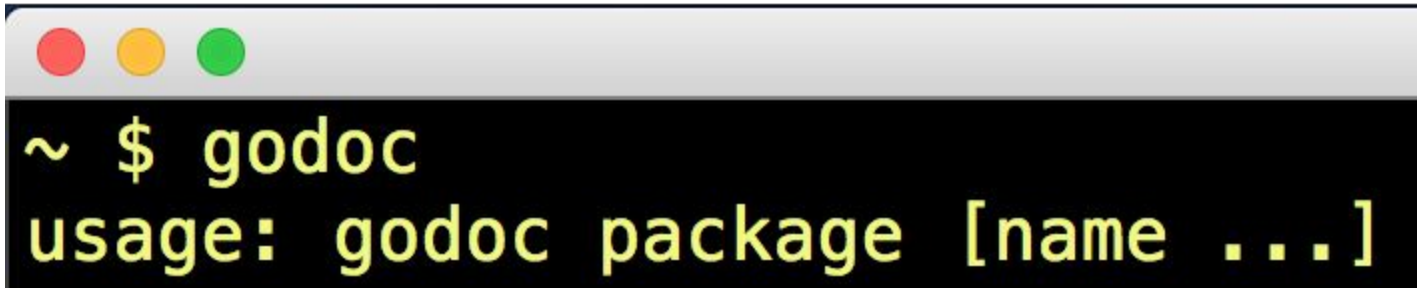
exercise

bring up the documentation for
reflect TypeOf
in terminal

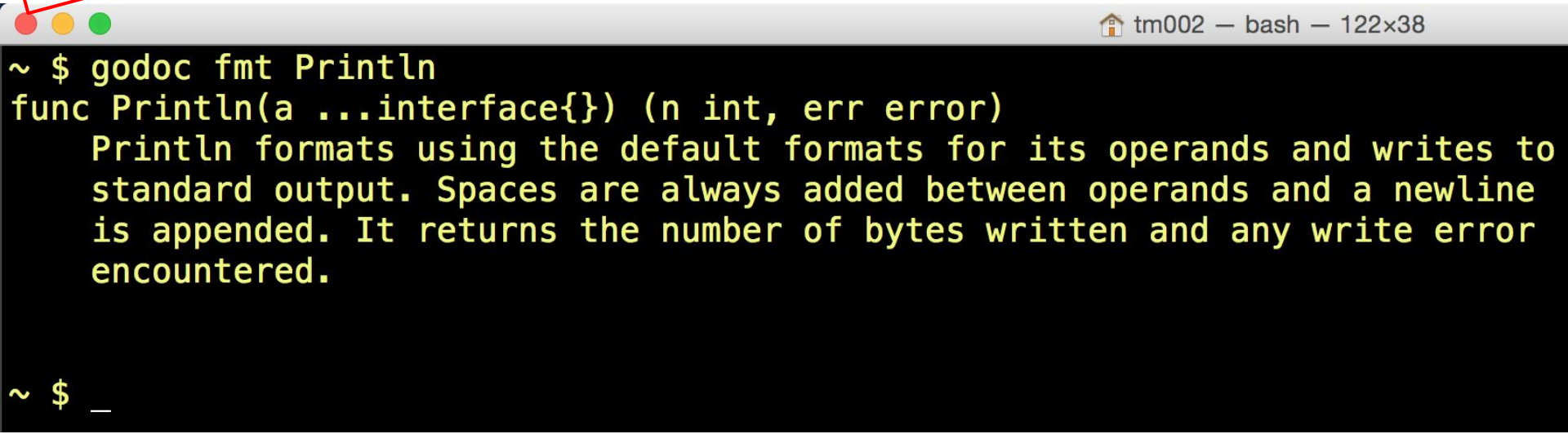

~ $ godoc reflect typeof
No match found.


case sensitive
tm002 — bash
~ $ godoc reflect TypeOf
type Type interface {
// Align returns the alignment in bytes of a value of
// this type when allocated in memory.
Align() int
// FieldAlign returns the alignment in bytes of a value of
// this type when used as a field in a struct.
FieldAlign() int
// Method returns the i'th method in the type's method set.

remember this syntax



here is another example
remember this one?



Review

- boolean
  - true
  - false
- numeric
  - int
  - uint
  - float64
- string
  - “”
  - ``
  - escape sequences
  - sequence of bytes
  - immutable
  - len
  - index access rune
    - “hello”[0]
  - slicing
    - “hello”[:4]
  - concatenate
    - +
- strconv
  - Atoi
  - Itoa
- conversion
  - T(x)
    - Type(variable)
    - string(‘a’)
    - string([]byte{'h', 'e', 'l', 'l', 'o'})
    - []byte(“Hello”)
    - float64(12)
    - int(12.1230123)
- assertion
  - used for interface types
- math package
- reflect package
  - TypeOf

Review Questions

static vs dynamic

- What is the difference between a **statically** and **dynamically** typed language?

static vs dynamic

- Is golang statically or dynamically typed?

static vs dynamic

- What type is like dynamic typing in golang?

rune

- In your words, provide a definition for **rune**.
- What is a **rune** in golang?
- Why is a **rune** an alias for int32 as opposed to int64 or uint32?

unsigned

- What is the difference between int & uint?

string index access

- Write a program that
  - create a string variable and assigns it some value
  - access a rune within the string variable using an index
    - prints this
  - slices the string
    - prints this

conversion

- Write a program that demonstrates these conversions
  - Type(variable)
  - string(‘a’)
  - string([]byte{'h', 'e', 'l', 'l', 'o'})
  - []byte(“Hello”)
  - float64(12)
  - int(12.1230123)

assertion

- Write a program that demonstrates assertion

int and uint

- What determines whether or not **int** and **uint** are 32 or 64 bytes?

type

- Create a program that displays a variables type using both
  - **TypeOf**
  - **%T** (a formatting verb used with Printf)

rand package

- We haven't talked about the rand package yet, however …
  - go read about the rand package on godoc.org
  - look at the example code for "magic eight-ball"
    - describe what **rand.Seed(42)** does
      - in your own words, what does "initialize the generator to a deterministic state" mean?
        - What does it mean to seed a random number generator?

          (you might need to google this)
      - what happens when you change the value from 42 to some other value?
        - can **rand.Seed(42)** take a float64?
    - describe what **rand.Intn(len(answers))** does