# Types & Expressions

02-201 / 02-601

# Types

# Variables beyond Integers

You can declare variables of several different built-in *types:*

| Type | Data | Examples |
|---|---|---|
| `int` | Positive or negative integers | 3,-200,40,42 |
| `uint` | Non-negative integers (u = unsigned) | 0, 3, 7, 11, 13 |
| `bool` | Holds **true** or **false** | **true** |
| `float64` | Real, floating point number | 3.14159, 12e-3, 0.23 |
| `complex128` | Complex number (real, imaginary) | |
| `string` | Holds a sequence of characters | “Hello, world” |

**Example variables declarations:**

```
var m uint = 10
var small bool = true
var big bool = m > 10
var e, pi float64 = 2.7182818285, 3.14
var name string = “Carl”
var root complex64 = 3 + 7i
```

# Literals

Explicit values for variables are called *literals*.

Integer literals: a sequence of digits 0…9

```
72
6402
000734
```

String literals: a sequence of characters between quotes “

```
“Hi there”
“😄”
“1+三=4”
“3.14159”
```

Unicode strings are supported.

**bool** (Boolean) literals: either **true** or **false**

```
true
false
```

Floating point (real) literals: a number with a “.” or “e”

```
7.
7.0
.32456
1.21212121
12E2
10E+3
11e-2
```

**Tip:** Use 7.0 and 0.32456 instead of 7. or .32456 (easier to read)

aEb = a × $10^b$

Imaginary literal: floating point literal with “i” after it

```
7.0i
7i
1e-5i
```

# Expressions and Variables Must Have the Same Type

```
var a float64 = 3.0          // ok!
var b float64 = “4.0”        // ERROR! “4.0” is a string
var c int = 3.0              // ERROR! 3.0 is not an int
var d string = 7000          // ERROR!
var e int = 2                // ok
var f uint = e               // ERROR! e is an int not a uint
var ok bool = 0              // ERROR! 0 is not a bool
var ok2 bool = e > 1         // ok: boolean expression
```

```
var scale int = 2                // ok
var t float64 = 2.3*scale        // ERROR! 2.3 is a float, scale is an integer
var t2 float64 = 2*scale         // ERROR! 2*scale is an integer
```

**Everything in a Go expression must have the same type.**

(this is different than C, C++, Java, which are more forgiving about types)

# Can Convert Between Types

Use *type(expression)* to covert *expression* to *type:*

```
var a float64 = 3.0          // ok!
var c int = int(3.0)         // ok: 3.0 converted to int
var g int = int(3.2)         // ERROR! can't convert 3.2 to int
var e int = 2                // ok
var f uint = uint(e)         // ok: e is converted to uint
var ok bool = bool(0)        // ERROR! can't covert ints to bools
```

Go really tries to avoid changing the value of a constant.

```
var time float64 = 7.2       // ok
var r int = time             // ERROR! time not an int
var round int = int(time)    // ok!!! round will equal 7
```

*You* know that `time` is 7.2, but Go doesn't know that, so it trusts you that you want to change `time` to an **int**.

When converting a floating point number to an int, Go will throw away the fractional part.

# Conversion Challenges

```
var a, b float64 = 7.6, -13.9
var c, d int = int(a), int(b)
```

Q: What values do c and d have?

Answer:

c == 7
d == -13

```
var u int = -70
var q uint = uint(u)
```

Q: What value does q have?

Answer: it depends on your computer, but on mine:

q == 18446744073709551546

What's going on here?

*More details later* →

- Go (and nearly all programming languages) represent negative integers in a format called "twos-complement"
- Converting a negative number to a **uint** simply reinterprets the bits used in twos-complement to represent a positive integer
- **This is almost never what you want.**
- **Lesson: converting a negative integer to a uint is probably a bug!**

# uint Challenge

```
var i uint = 10
for ; i >= 0; i = i - 1 {
    fmt.Println(i)
}
```

Q: How many times does this loop iterate?

Answer:
it never stops!

What's going on here?

- This is a common kind of bug
- **uint**s can't be negative:

```
var q uint = 0
q = q - 1
fmt.Println(q)
```

- This prints: 18446744073709551615
- q can't be negative, so "wraps around" to the largest possible **int**

# Variables Have Limited Range

| Type | Min | Max |
|---|---|---|
| `int` | -9223372036854775808 | 9223372036854775807 |
| `uint` | 0 | 18446744073709551615 |
| `float64` | -1.797693134862315708145274237317043567981e+308 | 1.797693134862315708145274237317043567981e+308 |

```
var i int = 9223372036854775807
fmt.Println(i+1)
```

Q: What does the above print?

Answer:

```
-9223372036854775808
```

i+1 is too big for an **int**, so it wraps around to the smallest possible **int**.

This is called *overflow* and its usually a bug!

**Lesson: if you have very big or very small numbers, you have to do something special.**

# Binary Numbers

**Base 10 (decimal) notation:**

$$
\begin{array}{rl}
 & 4\,2\,5\,6 \\
 & \downarrow\quad\downarrow\quad\downarrow\quad\downarrow \\
 & 6\text{x}10^{0} \\
 & 5\text{x}10^{1} \\
 & 2\text{x}10^{2} \\
+ & 4\text{x}10^{3} \\
\hline
 & 4\,2\,5\,6
\end{array}
$$

Computers store the numbers in binary because it has transistors that can encode 0 and 1 efficient

Each 0 and 1 is a *bit.*

Built-in number types each have a maximum number of bits.

**Base 2 (binary) notation:**

$$
\begin{array}{rl}
 & 1\,0\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0\,0 \\
 & \downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow \\
 & 0\text{x}2^{0} \\
 & 0\text{x}2^{1} \\
 & 0\text{x}2^{2} \\
 & 0\text{x}2^{3} \\
 & 0\text{x}2^{4} \\
 & 1\text{x}2^{5} \\
 & 0\text{x}2^{6} \\
 & 1\text{x}2^{7} \\
 & 0\text{x}2^{8} \\
 & 0\text{x}2^{9} \\
 & 0\text{x}2^{10} \\
 & 0\text{x}2^{11} \\
+ & 1\text{x}2^{12} \\
\hline
 & 4\,2\,5\,6 = \quad 1\,0\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0\,0
\end{array}
$$

# Variables with Different Ranges

| Type | Number of bits |
|---|---|
| `int` | 32 or 64 depending on your computer |
| `uint` | 32 or 64 depending on your computer (but always same size as int) |
| `int8 / uint8` | 8 |
| `int16 / uint16` | 16 |
| `int32 / uint32` | 32 |
| `int64 / uint64` | 64 |
| `float32` | 32 |
| `float64` | 64 |
| `complex64` | 32 for each of the real and imaginary parts |
| `complex128` | 64 for each of the real and imaginary parts |
| `byte` | another word for **`int8`** |
| `rune` | another word for **`int32`** |

**Tip:** use `int`, `float64`, and `complex128` unless you have memory limitations.

# Type Inference in var and := Statements

You can often omit specifying the type if Go can guess what type it should be:

```
var a = 3                  // a will be an int
var b = uint(a*a)          // b will be a uint
var d = “hi there”         // d will be a string
var ok = true              // ok will be a bool
var ok2 = e > 1            // ok2 will be a bool
var z = 3.0                // a will be a float64
q := 1e-23                 // q will be a float64
var theta = 3 + 2.0i       // theta will be a complex128
var t2 = 2*theta           // t2 will be complex128
result := f(a,b,d)         // will be the return type of f
```

Integer literals (3, -10, etc.) cause **int**s to be inferred

The larger float and complex size types (`float64`, `complex128`) are always used when inferring from literals of those types.

# The Lesson of Types

**Types in an expression must agree.**

**Be sure you don’t corrupt your data by converting to the wrong type.**

Everything else is basically details that you have to know to program, but that shouldn’t be forefront in your mind.

# Expressions & Operators

# Integer Operations

`a + b` addition

`a - b` subtraction

`a * b` multiplication

`-a` negation

`+a` doesn't do anything, but available for symmetry with -

`a / b` ***integer*** division: 2/3 = 0; 10/3 = 3; -10/3 = -3
results are truncated toward 0

If q = x / y and r = x % y then
x = q*y + r and |r| < |y|

`a % b` *modulus* (aka remainder):
13 % 2 = 1; 10 % 2 = 0; 10 % 3 = 1
-10 % 3 = -1 (since -10 = 3*(-3) - 1)

# Increment and Decrement Statements

Adding and subtracting 1 is so common there is a special notation for it:

`a++` is the same as `a = a + 1`

`a--` is the same as `a = a - 1`

This is particularly useful in **for** loops:

```
for i := 1; i <= n; i++ {
  // body of for loop
}
```

# Float and Complex Operators

```
a + b   a - b   a * b   -a   +a
```

```
a / b
```

*floating point* division:

2.0/3.0 = 0.6666666666666666

10.0/3.0 = 3.3333333333333335

10.0/3.0 = -3.3333333333333335

results are of limited precision

b can’t be 0

# Boolean Operators

`a && b` **true** if and only if a and b are both **true**

`a || b` **true** if and only if one of a or b is **true**

# Comparison Operators

`a < b` `a > b` `a == b` equals

`a <= b` `a >= b` `a != b` not equals

# Operator Precedence

```
x * y + z
```

= (x*y) + z

applied first →

| Precedence | Operator |
|---|---|
| 5 | * / % |
| 4 | + - |
| 3 | == != < <= > >= |
| 2 | && |
| 1 | \|\| |

← applied last

Use () to group operators and change the order they are applied: x*(y+z)

**Tip: don't** remember the order of operations: always use () to make the order explicit.

# Example Expressions

```
a + b / 3 + 2
```

```
(a+b) / 3 + 2
```

```
-a*(3+c - d)
```

# Strings & The Motivation for Types

# String Operator

`s1 + s2` concatenation

Places one string after another:

```
“hi” + “there” == “hithere”
“what’s “ + “up” + “ doc?” == “what’s up doc?”
```

Question: What’s the value of a?

```
a := “hi” + 42
```

Answer: this is an error, since “hi” has type **string** and 3 has type **int**.

# String Representations

The name “string” is meant to suggest a sequence of characters strung together.


Characters are stored as binary numbers (as is *all* data in the computer).

The type of a variable tells your program how to interpret those numbers as data that makes sense in the world.

**int** means “interpret these binary digits as an integer”
**float64** means “interpret these binary digits as a real number”
**string** means “interpret these binary digits as a sequence of characters”

Question: What’s the value of a?

```
a := “hi” + string(42)
```

# ASCII Chart
# (You don’t need to know these numbers)

| Binary | Dec | Glyph |
|---|---|---|
| 110 0000 | 96 | ` |
| 110 0001 | 97 | a |
| 110 0010 | 98 | b |
| 110 0011 | 99 | c |
| 110 0100 | 100 | d |
| 110 0101 | 101 | e |
| 110 0110 | 102 | f |
| 110 0111 | 103 | g |
| 110 1000 | 104 | h |
| 110 1001 | 105 | i |
| 110 1010 | 106 | j |
| 110 1011 | 107 | k |
| 110 1100 | 108 | l |
| 110 1101 | 109 | m |
| 110 1110 | 110 | n |
| 110 1111 | 111 | o |
| 111 0000 | 112 | p |
| 111 0001 | 113 | q |
| 111 0010 | 114 | r |
| 111 0011 | 115 | s |
| 111 0100 | 116 | t |
| 111 0101 | 117 | u |
| 111 0110 | 118 | v |
| 111 0111 | 119 | w |
| 111 1000 | 120 | x |
| 111 1001 | 121 | y |
| 111 1010 | 122 | z |
| 111 1011 | 123 | { |
| 111 1100 | 124 | \| |
| 111 1101 | 125 | } |
| 111 1110 | 126 | ~ |

| Binary | Dec | Glyph |
|---|---|---|
| 100 0000 | 64 | @ |
| 100 0001 | 65 | A |
| 100 0010 | 66 | B |
| 100 0011 | 67 | C |
| 100 0100 | 68 | D |
| 100 0101 | 69 | E |
| 100 0110 | 70 | F |
| 100 0111 | 71 | G |
| 100 1000 | 72 | H |
| 100 1001 | 73 | I |
| 100 1010 | 74 | J |
| 100 1011 | 75 | K |
| 100 1100 | 76 | L |
| 100 1101 | 77 | M |
| 100 1110 | 78 | N |
| 100 1111 | 79 | O |
| 101 0000 | 80 | P |
| 101 0001 | 81 | Q |
| 101 0010 | 82 | R |
| 101 0011 | 83 | S |
| 101 0100 | 84 | T |
| 101 0101 | 85 | U |
| 101 0110 | 86 | V |
| 101 0111 | 87 | W |
| 101 1000 | 88 | X |
| 101 1001 | 89 | Y |
| 101 1010 | 90 | Z |
| 101 1011 | 91 | [ |
| 101 1100 | 92 | \ |
| 101 1101 | 93 | ] |
| 101 1110 | 94 | ^ |
| 101 1111 | 95 | _ |

| Binary | Dec | Glyph |
|---|---|---|
| 010 0000 | 32 | (space) |
| 010 0001 | 33 | ! |
| 010 0010 | 34 | " |
| 010 0011 | 35 | # |
| 010 0100 | 36 | $ |
| 010 0101 | 37 | % |
| 010 0110 | 38 | & |
| 010 0111 | 39 | ' |
| 010 1000 | 40 | ( |
| 010 1001 | 41 | ) |
| 010 1010 | 42 | * |
| 010 1011 | 43 | + |
| 010 1100 | 44 | , |
| 010 1101 | 45 | - |
| 010 1110 | 46 | . |
| 010 1111 | 47 | / |
| 011 0000 | 48 | 0 |
| 011 0001 | 49 | 1 |
| 011 0010 | 50 | 2 |
| 011 0011 | 51 | 3 |
| 011 0100 | 52 | 4 |
| 011 0101 | 53 | 5 |
| 011 0110 | 54 | 6 |
| 011 0111 | 55 | 7 |
| 011 1000 | 56 | 8 |
| 011 1001 | 57 | 9 |
| 011 1010 | 58 | : |
| 011 1011 | 59 | ; |
| 011 1100 | 60 | < |
| 011 1101 | 61 | = |
| 011 1110 | 62 | > |
| 011 1111 | 63 | ? |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Logical interpretation: | **H** | **e** | **l** | **l** | **o** | **,** | | **W** | **o** | **r** | **l** | **d** | **!** |
| Representation in the computer: | **72** | **101** | **108** | **108** | **111** | **44** | **32** | **87** | **111** | **114** | **108** | **100** | **33** |

# Packages

# Packages

- Packages are collections of functions you can use in your program.

- Go provides many built-in packages →

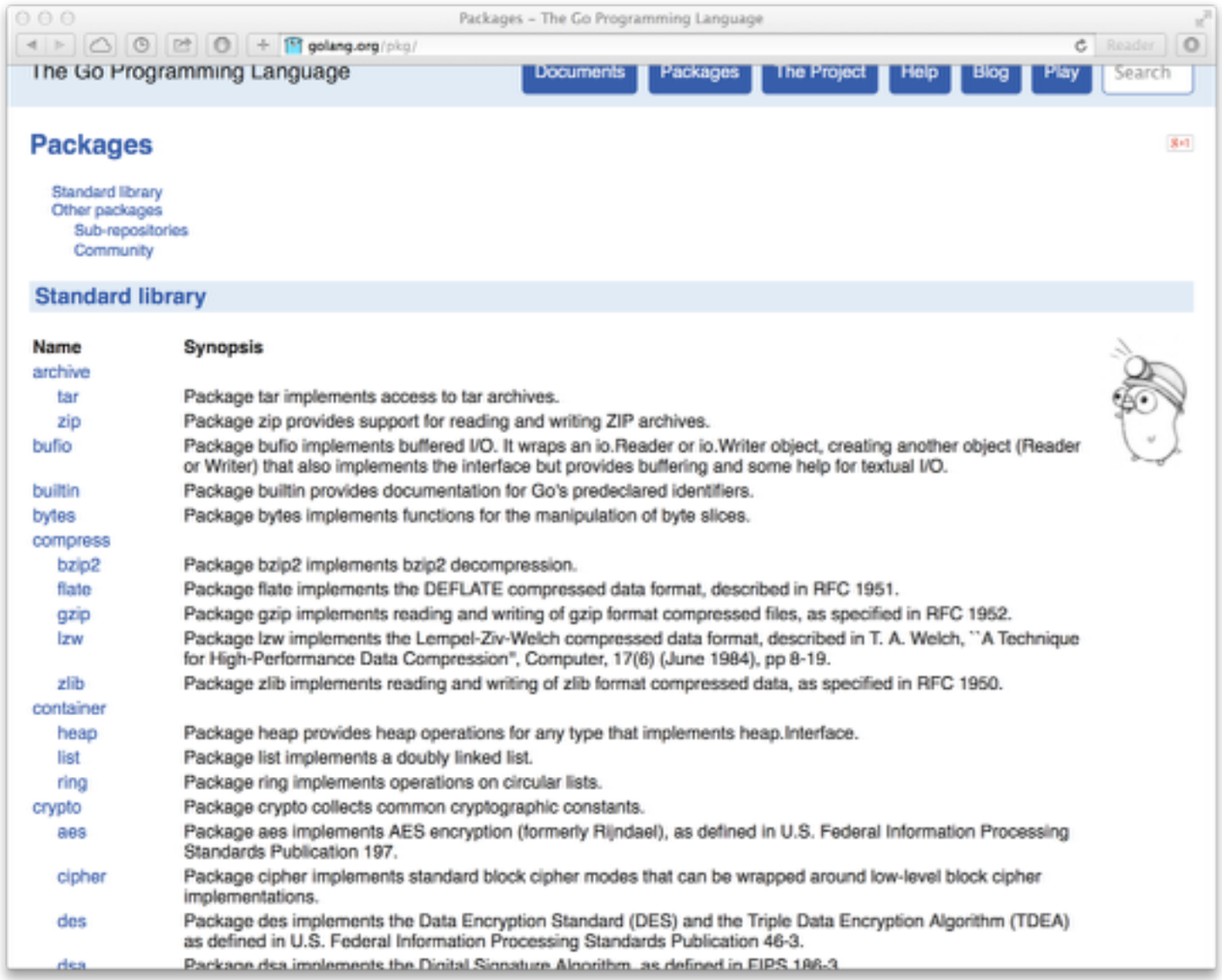
- Enable the the use of a package with:

```
import "packageName"
```

  at the top of your program.

- Get a list of built-in packages at: http://golang.org/pkg/

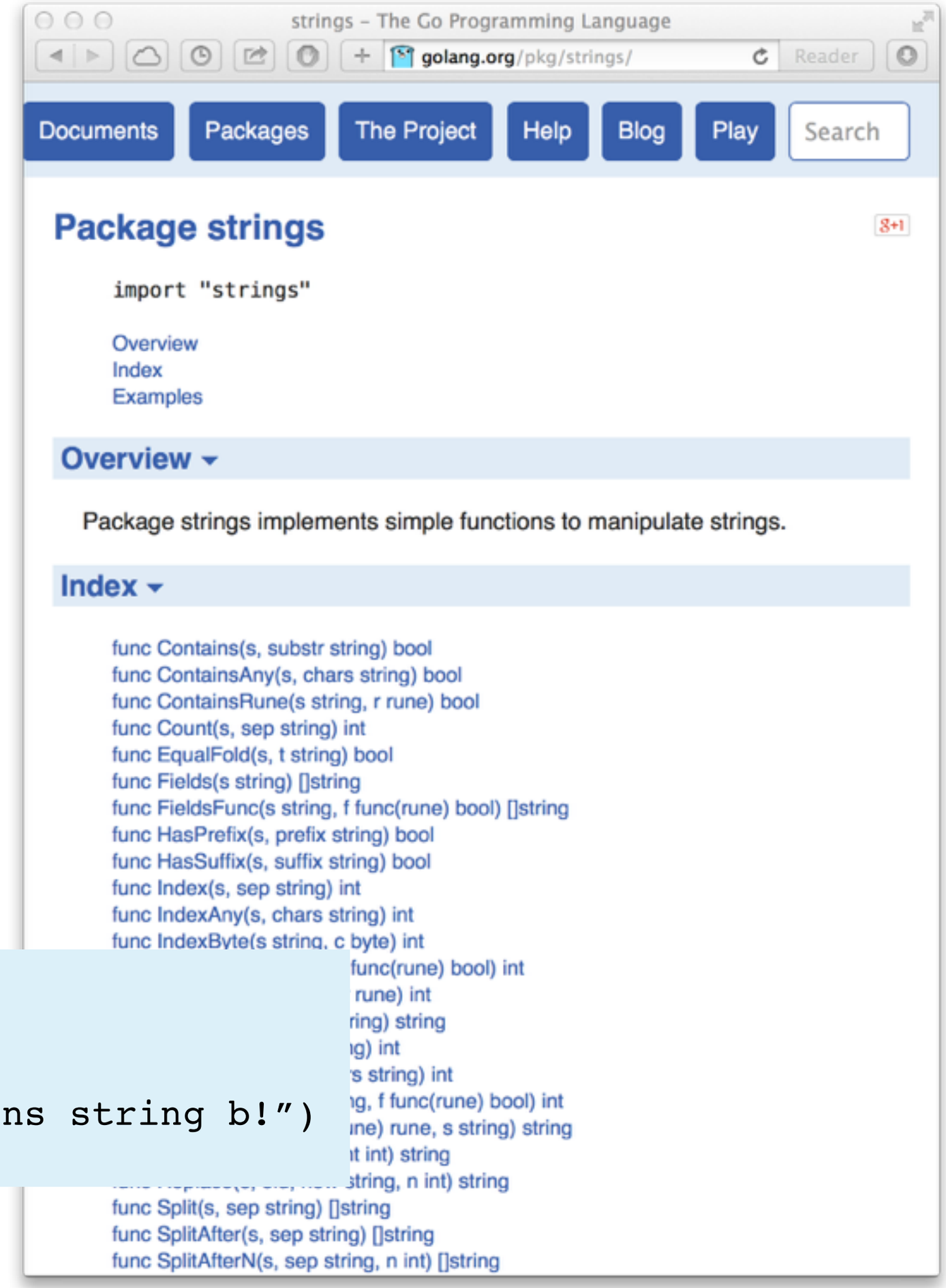
- `fmt` package provides the `fmt.Print` and `fmt.Println` functions we've used a lot.



# String Package

- Lets you manipulate strings.
- A very large part of programming in practice is looking up how to use functions in existing packages.
- This example tests whether one string has another as a substring:

```
var a = “hi, there”
var b = “the”
if strings.Contains(a, b) {
    fmt.Println(“String a contains string b!”)
}
```



# strconv Package

Back to our problem: how do we get this:

```
a := “hi” + string(42)
```

to do what we want?

Inside of the strconv package, there are these functions:

```
func FormatBool(b bool) string
func FormatFloat(f float64, fmt byte, prec, bitSize int) string
func FormatInt(i int64, base int) string
func FormatUint(i uint64, base int) string
func Itoa(i int) string
```

Takes in an **int64** and returns a **string**

FormatInt also allows you to set the base (10, etc.) of the representation.

Using the Itoa() function gives us our desired result:

```
a := “hi” + strconv.Itoa(42)
```

# math Package

Might find:

```
math.Abs()
math.Pow10()
```

functions useful for the `KthDigit` question on homework 1.

# Types & Expressions Summary

- Every variable has a type that tells Go how to interpret the bits that represent that variable.
- In Go, everything in an expression must have the same type.
- You can convert between types by using the type name like a function: **int**(myFloatVar).
- Consistent with other types: **string**(103) reinterprets the number 103 as a string. It does not turn 103 into a string “103” of the decimal representation of 103.
- Packages provide lots of useful pre-defined functions, one of which is to convert numbers into strings (and vice versa).

# Go Summary

<u>90% of programming is the combination of these things:</u>

functions: basic building block: define new “things” the computer can do

variables: units of data that your program can manipulate

types: tell Go what kind of data is in each variable

**if…else**: select statements to execute based on some condition

**for**: repeat statements while some condition is true