Lecture 9: Lists

Terminology: Go uses a non-standard term <u>slice</u> to refer to what we are calling <u>lists</u>. Others use the term <u>array</u> for the same concept. Unfortunately, Go uses <u>array</u> for a related, but slightly different, thing. The bottom line: Go uses the word <u>slice</u> for what we are calling <u>lists</u>.

Lists store lists of variables

For example, a list might be used to store:

- A list of filenames
- A list of prime numbers
- A column of data from a spreadsheet
- A collection of DNA sequences
- Factors of a number
- etc.

3	12	3	3	7	8	10	-2	30	6	11	11	11	32	64	80	99	-1	0	12

Declaring variables that hold lists

To delcare a list you preceed the type with `[ ]` :

```
1  var a []int     // a list of integers
2  var b []string  // a list of strings
3  var c []float64 // a list of floats
```

For example, `a` is a list of integers. However, we haven't specified *how many* integers are in this list. In addition, when declared, lists have the special value `nil` , which means that they cannot be used until we say how many things will be in the list.

To start to use the list, we have to `make` it:

```
1  var a []int
2  a = make([]int, 10) // a is now an list of length 10
```

As always, we can use Go's type inference to avoid having to specify the list type twice:

```
1  var a = make([]int, 10)
2  b := make([]float64, 300)
3  c := 124
4  d := make([]float64, c) // d is a list of length 124
```

Accessing list elements

Each item in the list is called an element. List elements can be accessed by putting their index between `[ ]` following the list name:

```
1  a := make([]int, 10)
2  b := make([]string, 100)
3  c := make([]float64, 500)
4
5  fmt.Println(a[7],a[8])
```

Pictorially, we have:

list elements:	13	18	-2	10	11	10	-22	8	8	7	-30	-33	-22	12	99	98	97	6	-3	2
index into list:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

`x[i]` can appear on left-hand side of assignment to set a value:

```
1  a[0] = 10
2  b[30] = "hi there"
3  i := 12 + 2
4  c[i] = 3.1
5  c[2*i] = c[i]
```

The length of a list can be found with `len(x)`, where `x` is a list variable.

List indices start at 0! The first element is `x[0]`.

The last element is at index `len(x) - 1`.

It's an error to try to access elements past the end of the list:

```
1   var d []int = make([100]int)
2   d[0] = 2              // ok
3   d[99] = 70            // ok
4   var j int = 100
5   fmt.Println(d[j])    // ERROR!
6   d[len(d)-1] = 3      // OK
7   d[len(d)] = 3    // ERROR!
8   d[-60] = 7       // ERROR!
```

These errors may only be caught when your program runs.

Example: Sieve of Eratosthenes

The "Sieve of Eratosthenes" is a very old algorithm for finding prime numbers:

```
1   func primeSieve(isComposite []bool) {
2       var biggestPrime = 2 // will hold the biggest prime found so far
3       for biggestPrime < len(isComposite) {
4           // knock out all multiples of biggestPrime
5           for i := 2*biggestPrime; i < len(isComposite); i += biggestPrime {
6               isComposite[i] = true
7           }
8           // find the next biggest non-composite number
9           biggestPrime++
10          for biggestPrime < len(isComposite) && isComposite[biggestPrime] {
11              biggestPrime++
12          }
13      }
14  }
```

Why does this work?

At start of outer for loop:

isComposite:	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
index into list:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

↑
biggestPrime

First inner for loop sets all multiples of biggestPrime to be TRUE:

F	F	F	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

↑
biggestPrime

Second inner for loop increments biggestPrime until it finds a non-composite number:

F	F	F	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

↑
biggestPrime

Next time through the outer loop, multiples of 3 will be marked as composite, etc.

Aside: Shortcut `&&` and `||`

Consider this loop from `primeSieve()`:

```
for biggestPrime < len(isComposite) && isComposite[biggestPrime] {
    biggestPrime++
}
```

What happens when `biggestPrime == len(isComposite)`?

- The green (first) condition is false
- The red (second) condition is an ERROR

So does this program have a bug? No:

The `&&` and `||` operators work from left to right and stop once their truth value can be determined.

Once the green condition is false, there's no way for the whole expression to be true, so in that case, the red condition is never evaluated.

Calling the Sieve

```go
func main() {
    var composites []bool = make([]bool, 100000000)
    primeSieve(composites)

    for i := 0; i < len(composites); i++ {
        if !composites[i] && i >= 2 {
            primeCount++
            fmt.Println("Number of primes ≤", i, "is", primeCount)
        }
    }
}
```
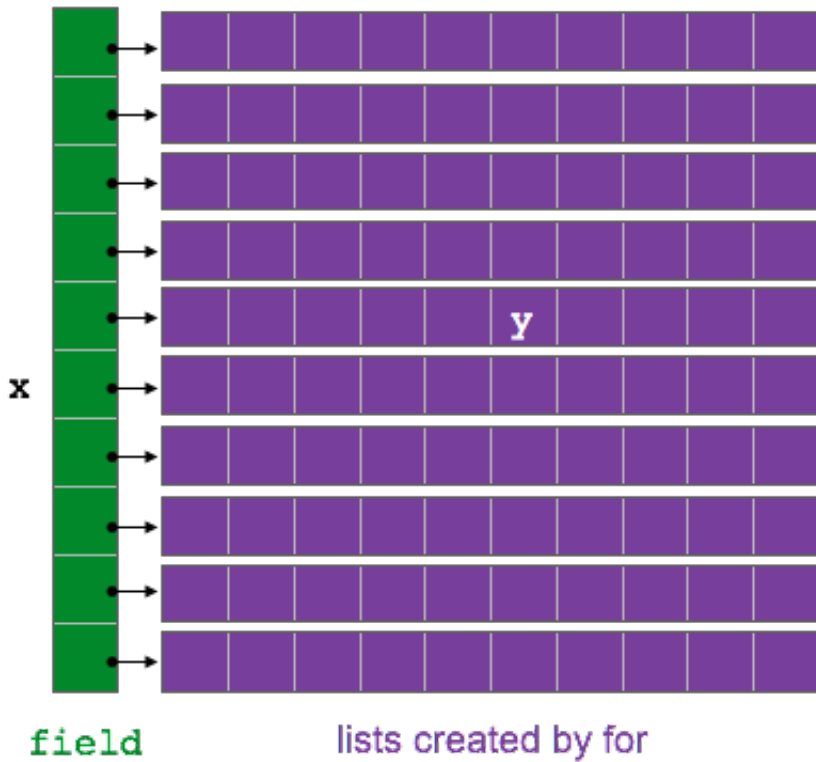
Example: Self-avoiding random walks

Problem: Simulate a random walk on an n-by-n chessboard but **don't allow the walk to visit the same square twice.**

Need to keep track of where the walk has already visited.

Two-dimenstional lists

A 2-D list is just a list of lists:

field   lists created by for

You declare them as you might expect:

```
1  var field [][]bool = make([][]bool, n)
```

The `make` above creates the list of lists (green), but does not create the actual inner lists (purple). For that, we must write our own loop:

```
1  for i := 0; i < n; i++ {
2      field[i] = make([]bool, n)
3  }
```

> **Test yourself!** Why did we use `[]bool` above instead of `[][]bool` ?

We can now use `field` as a 2-dimensional array:

```
1  var x, y = len(field)/2, len(field)/2
2  field[x][y] = true
```

> **Test yourself!** How would you create a 3-dimensional array? What about an `i` dimensional array, where `i` is a variable in your program.

2-d self avoiding random walks

The lines with `***` are where we mark a square visited.

```go
func selfAvoidingRandomWalk(n, steps int) {
    var field [][]bool = make([][]bool, n)
    for row := range field {
        field[row] = make([]bool, n)
    }
    var x, y = len(field)/2, len(field)/2

    field[x][y] = true // ***
    fmt.Println(x,y)

    for i := 0; i < steps; i++ {
        // repeat until field is empty
        xnext, ynext := x, y
        for field[xnext][ynext] {
            xnext,ynext = randStep(x, y, len(field))
        }
        x, y = xnext, ynext
        field[x][y] = true // ***
        fmt.Println(x,y)
    }
}

func randDelta() int {
    return (rand.Int() % 3) - 1
}

func inField(coord, n int) bool {
    return coord >= 0 && coord < n
}

func randStep(x,y,n int) (int, int) {
    var nx, ny int = x, y
    for (nx == x && ny == y) || !inField(nx,n) || !inField(ny,n) {
        nx = x+randDelta()
        ny = y+randDelta()
    }
    return nx, ny
}
```

Notice that this is yet a 3rd way to structure our random walk code.

A little bug

BUG: What if the walk gets stuck and can't move? What will happen in the above code then?

The solution is to add some code to check if the walk is stuck, and in that case to stop trying to move:

```
1   // returns true if we are stuck
2   func stuck(x,y int, field [][]bool) bool {
3       for dx := -1; dx <= 1; dx++ {
4           for dy := -1; dy <= 1; dy++ {
5               nx, ny := x+dx, y+dy
6               if inField(nx, n) && inField(ny, n) && !field[nx][ny] {
7                   return false
8               }
9           }
10      }
11      return true
12  }
```

We then add:

```
1   if stuck(x,y,field) {
2       return
3   }
```

after line 11 in our self-avoiding random walk program.

Strings: a special kind of list

Strings work like arrays of uint8s in some ways:

- You can access elements of string `s` with `s[i]` .
- You can get their length with `len(s)` .

However:

- You cannot modify a the characters of a string once it has been created. So `s[i] = 'a'` is not allowed for strings.

Example string manipulation:

Here is a function that returns the complement of a DNA character:

```
// Complement computes the reverse complement of a
// single given nucleotide. Ns become Ts as if they
// were As. Any other character induces a panic.
func Complement(c byte) byte {
    if c == 'A' { return 'T' }
    if c == 'C' { return 'G' }
    if c == 'G' { return 'C' }
    if c == 'T' { return 'A' }

    panic(fmt.Errorf("Bad character: %s!", string(c)))
}
```

A `panic` is a special function that terminates your program with an error. You should use it when something occurs that really should not happen.

We can use the `Complement` function now to compute the reverse complement of a DNA sequence:

```
// reverseComplement() returns the reverse
// complement of the given string
func reverseComplement(r string) string {
    s := make([]byte, len(r))
    for i := 0; i < len(r); i++ {
        s[len(r)-i-1] = Complement(r[i])
    }
    return string(s)
}
```

Note that `string(s)` turns a byte array into a `string`.

List literals

You can specify a list directly in your program using:

```
[]float64{3.2, -30, 84, 62}
[]int{1,2,3,6,7,8}
```

This is useful if you have a fixed, short list of data.

`for ... range`: making iterating through lists easier

The pattern:

```go
for i:= 0; i < len(list); i++ {
    // use list[i]
}
```

is so common that Go provides a special syntax for it:

```go
for i, elementI := range list {
    // here elementI is equal to list[i]
}
```

This will loop through all the elements of `list` in order, setting `elementI` to each one in turn. (You can use whatever names you want for `i` and `elementI`.)

Another example:

```go
var max, pos int
for j, v := range list {
    if j == 0 || v > max {
        max = v
        pos = j
    }
}
fmt.Println("Max value is", max, "at", pos)
```

If you only need the indices and not the elements you can write:

```go
for i := range list {
    // i will go from 0 to len(list)-1
}
```

For example:

```
1   for i := range list {
2       list[i] = -(i - 6)*(i-6)
3   }
4   fmt.Println(list)
```

If you only need the elements, and not their indices, you can write:

```
1   for _, elementI := range list {
2       // elementI will be equal to each list element in turn
3   }
```

The _ is a single underscore character and is a special variable name. It is called the blank identifier and can
be used anyplace you have to provide a variable name, but don't actual care about the variable.

For example:

```
1   func sum(A [10]int) {
2       var result int
3       for _, val := range A {
4           result = result + val
5       }
6       return result
7   }
```

Appending elements to the end of a list

We can grow lists by adding things to the end. This is done using the append function:

```
1   s := make([]int, 10)
2   s = append(s, 5)
```

| S:	0	0	0	0	0	0	0	0	0	0	5
	0	1	2	3	4	5	6	7	8	9	10

After **append**

Note the syntax is somewhat redundant: s = append(s, 5) . This is required for a technical reason. The

important thing to remember is that you have to assign the return value of `append` back to the list you are appending to.

We can now re-write our use of the prime sieve:

```
func main() {
    var composites []bool = make([]bool, 100000000)
    primeSieve(composites)
    var primeCount int = 0
    var primesList []int = make([]int, 0)
    for i, isComp := range composites {
        if !isComp && i >= 2 {
            primeCount++
            fmt.Println("Number of primes ≤", i, "is", primeCount)
            primesList = append(primesList, i)
        }
    }
}
```

Another example: find the points that fall inside of a given rectangle.
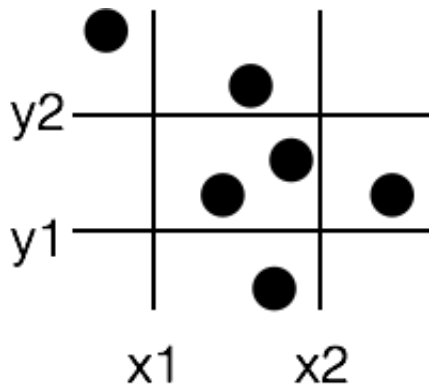
```go
// take a box and list of 2D points and return the 2D points that lie in the box
func pointsInBox(
    x1,y1,x2,y2 float64,
    xs, ys []float64
) ([]float64, []float64) {

    var xout = make([]float64, 0)
    var yout = make([]float64, 0)

    for i := range xs {
        if x1 <= xs[i] && xs[i] <= x2 && y1 <= ys[i] && ys[i] <= y2 {
            xout = append(xout, xs[i])
            yout = append(yout, ys[i])
        }
    }
    return xout, yout
}

func main() {
    var x = []float64{-1, 3.2, 7.8, -2.45}
    var y = []float64{-2, -4.0, 3.14, 2.7}

    xlist, ylist := pointsInBox(-5,-5,5,5, x, y)

    for i := range xlist {
        fmt.Println(xlist[i], ylist[i])
    }
}
```

Summary

Lists store collections of variables of the same type.

The length of a list can be found with: `len(name)`

`name[i]` is a variable that is the ith element of the list.

`name[0]` is the first element of the list.

You have to explicitly write code to create 2-D (or 3-D, etc.) lists.

Glossary

- blank identifier: The special variable name `_` that can be used when you don't care about a variable, but have to provide one.
- list: A sequence of elements of the same type.
- element: A single item in a list.
- slice: Go's term for what we are calling lists.
- array: Some programming languages use the term array for we are calling lists.