Learning Objectives

Learners will be able to...

- **Declare and initialize variables, constants, arrays and slices with the appropriate data types**
- **Control program flow using loops, conditions, and functions**
- **Handle errors**

info

Make Sure You Know

Python or other modern programming language.

Data Types and Variables

Unlike dynamically typed programming languages (e.g. Python, JavaScript), Go is a **statically typed language**. That means that when declaring variables and constants, you must specify the type.

Data Types

The basic Go types are summarized in the table below.
| Data Type | Examples | Description |
|————|————-|————-|
| `bool` | `true`
`false` | Represents the set of Boolean truth values. |
| `string` | `"text"`
`` `text` ``
`` `multiline
string` `` | You can use " or ` to denote a string literal.
Backticks are used for multi-line strings.
There is a function `len` to get string length.
Characters can be accessed by index (i.e. `s[i]`).
Strings are immutable. |
| `int` | `6` | Signed (both positive and negative) integers either 32 bit or 64 bit based on the system. |
| `float64` | `123.456` | Decimal point numbers |

Above is only a subset of the most common numeric data types.
▼ **Click to see all of the numeric types.**

Data Type	Description
uint	32 bits on 32-bit systems and 64 bits on 64-bit systems
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)
int	32 bits on 32-bit systems and 64 bits on 64-bit systems

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

When you need an integer value, use `int` unless you have a specific reason to use a sized or unsigned integer type.

info

Creating your own `types`

The `type` operator allows you to define a named type based on an already existing data type.

For example:

```
type meter int
```

In the future, we can easily change this type, for example, to `float32`, which can be very useful when changing the basic requirements for the application:

```
type meter float32
```

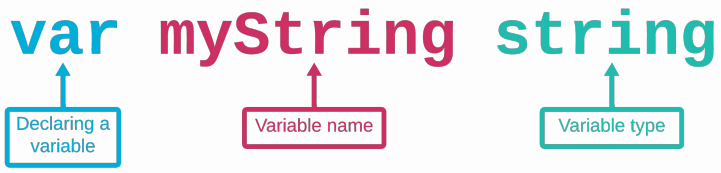
`type` can also create structures containing fields from primitive data types, other structures, and named types.

```
type Contacts struct {
    Phone int
    Email string
}

type User struct {
    Name     string
    Contacts Contacts
}
```

Declaring and Assigning Variables

To declare a variable, the `var` keyword is used, followed by the name of the variable and its type.



declare a variable by using the `var` keyword followed by the

name of the variable followed by its type

```
var variableName float64
// or
var VariableName bool
```

xdiscipline

Code Style: Variable names

Go uses both CamelCase and lowerCamelCase when naming variables and constants.
* **Non-exported variables** should begin with a **lowercase** letter
* **Exported variables** should begin with a **CAPITAL** letter.

Abbreviations in variable names (for example, URL, ID) must be in the same case (e.g. `HTTPResponse` or `httpResponse`).

As with other programming languages, variable names must not be a keyword (e.g. `break`, `case`, `const`, `continue`, `default`).

Try declaring a `string` variable called `greetingText` in the file to the left:

```
var greetingText string
```

You can declare several variables of the same type, separated by commas:

```
var i, j, k int
```

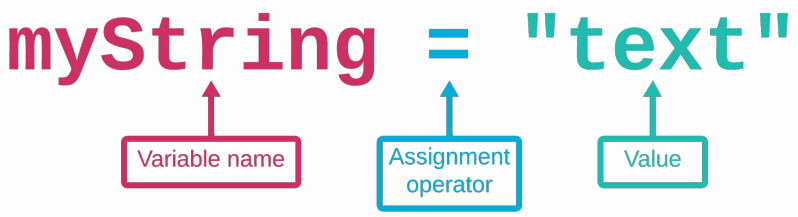
xdiscipline

Code Style: Indentation

Similar to other languages, when putting statements inside a structure such as a loop or function, the statements are indented.

Unlike Python, indentation is optional.

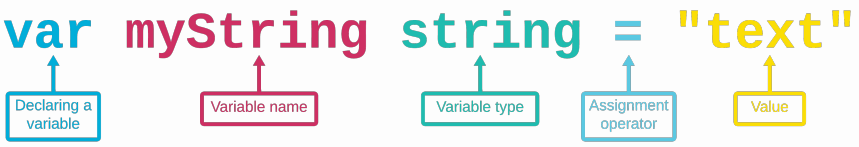
Once a variable has been declared, it can be assigned a value:



Try assigning a value to `greetingText` in the file to the left:

```
greetingText = "Hello Go!"
```

When declaring a variable, you can assign a value to it immediately:



Try combining your declaration and assignment of `greetingText` into one statement:

```
var greetingText string = "Hello Go!"
```

You can declare and initialize multiple variables at once:

```
var (
    course string = "Go For Python"
    modules int = 9
)
```

definition

Constants

Unlike variables, the value of constants cannot be changed. When declaring a constant, it must be initialized.

A constant is defined by the keyword `const`. For example:

```
const daysInWeek int = 7
```

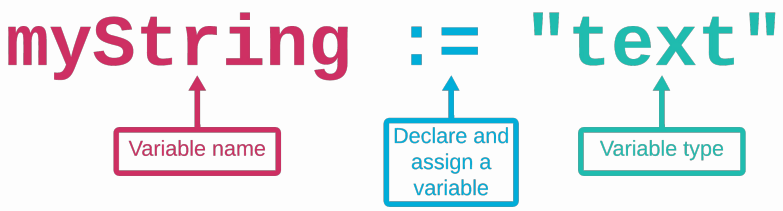
Similar to variables, you can declare and initialize multiple constants at once:

```
const (
    daysInWeek int = 7
    daysInYear int = 365
)
```

If you initialize a constant at declaration, you can let the type be inferred:

```
const daysInWeek, daysInYear = 7, 365
```

In Go, there is syntactic sugar or a way to shorten the definition and assignment of a variable:



The data type for the variable is automatically inferred. For example:

```
course := "Go For Developers"
```

Type conversion

Type conversion in Go may appear explicitly in the source, or it may be implied by the context in which an expression appears.

An **explicit conversion** is an expression of the form $T(x)$ where T is a type and x is an expression that can be converted to type T:

```
import (
  "fmt";
  "reflect"
)

func main() {
  numFloat := 1.123
  fmt.Println(reflect.TypeOf(numFloat)) // float64
  numInt := int(numFloat)
  fmt.Println(numInt) // 1
  fmt.Println(reflect.TypeOf(numInt)) // int

  str := "Codio"
  fmt.Println(reflect.TypeOf(str)) //string
  sliceRunes := []rune(str)
  fmt.Println(sliceRunes) // [67 111 100 105 111]
  fmt.Println(reflect.TypeOf(sliceRunes)) // []int32
}
```

You can only perform type casting between compatible data types. For example, you cannot convert an integer to a string value directly using the cast operator. Instead, you need to use the `strconv` package to convert an integer to a string:

```
import (
  "fmt";
  "reflect";
  "strconv"
)

func main() {
  num := 123
  numString := strconv.Itoa(num)
  fmt.Println(numString) // 123
  fmt.Println(reflect.TypeOf(numString)) // string
}
```

Operators

Arithmetic operators apply to numeric values and produce a result of the same type as the first operand. The + operator also applies to strings.

Below is a table of arithmetic operators with a description and type of operands to which they apply.
Operator | Description | Compatible Types
:——:|———————|———
+ |sum |integers, floats, complex values, strings
- |difference |integers, floats, complex values
* |product |integers, floats, complex values
/ |quotient |integers, floats, complex values
% |remainder |integers

Try it out on the left:

```
number1 := int(3)
number2 := int(4)
sum := number1 + number2
fmt.Println(strconv.Itoa(sum))
```

important

Types and Arithmetic Operators

As Go is strongly typed, it does not allow mixing of numeric types in expressions.

See the error message if you change the example above to:
`number2 := int64(4)`

Bitwise operations are performed on individual digits of a number's binary representation.
Operator | Description | Compatible Types
:——:|———————|———
& |bitwise AND |integers
| |bitwise OR |integers
^ |bitwise XOR |integers
&^ |bit clear (AND NOT) |integers

Try it out on the left:

```
num1 := int64(3)
fmt.Println(strconv.FormatInt(num1, 2)) // 11
num2 := int64(4)
fmt.Println(strconv.FormatInt(num2, 2)) // 100
res := num1 & num2
fmt.Println(strconv.FormatInt(res, 2)) // 0
res = num1 | num2
fmt.Println(strconv.FormatInt(res, 2)) // 111
```

Shift operations allow you to shift the binary representation of a number several places to the right or left. Shift operations apply only to integer operands.
Operator | Description | Compatible Types
:——:|——————|———
<< |left shift |integer << integer >= 0
>> |right shift |integer << integer >= 0

Try it out on the left:

```
num := int64(5)
fmt.Println(strconv.FormatInt(num, 2)) // 101
res := num << 2
fmt.Println(strconv.FormatInt(res, 2)) // 10100
res = num >> 2
fmt.Println(strconv.FormatInt(res, 2)) // 1
```

important

Order of operations

There are five precedence levels for binary operators:

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

Within the same level of precedence, operators are evaluated from left to right.

Postfix increment (`i++`) and **postfix decrement** (`i--`) increment or decrement a variable by 1, respectively:

```
var num int = 3
num++
fmt.Println(num) // 4
num--
fmt.Println(num) // 3
```

Logical operators

Comparison operators compare two operands and yield an untyped boolean value.

Operator	Description
==	equal
!=	not equal
<	less
<=	less or equal
>	greater
>=	greater or equal

In any comparison, the first operand must be assignable to the type of the second operand, or vice versa.

Try it out on the left:

```
import "fmt"

func main() {
  var num int = 3
  var num2 int = 5

  fmt.Println(num < num2)
  fmt.Println(num >= num2)
  fmt.Println(num != num2)
}
```

The equality operators == and != apply to operands of comparable types. The ordering operators <, <=, >, and >= apply to operands of ordered types.

Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally.

Operator	Description
&&	conditional AND
||	conditional OR
!	NOT

Try it out on the left:

```
fmt.Println(true && true)
fmt.Println(true || false)
fmt.Println(!false)
```

Arrays and Slices

Go includes statically-sized **Arrays** and dynamically-sized **Slices**.

Arrays

Arrays represent a sequence of elements of a certain type. An array is defined using the following syntax:

```
var array_name [number_of_elements]type_of_elements
```

For example:

```
var array [3]int

fmt.Println(array)
```

Note that when an array is declared, it is also initialized with the default value of the stated type.

You can set the individual elements of an array:

```
array[0] = 1
array[1] = 2
array[2] = 3

fmt.Println(array)
```

Like most programming languages, Go starts with index 0 so a length 3 array has indices 0 to 2.

You can use syntactic sugar to declare and instantiate an array all at once:

```
numbers := [10]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
// or
//numbers := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
fmt.Println(numbers)
```

When you declare an array all at once, the length is optional and can be replaced with `...`.

You can access an element with an integer index as follows:

```
fmt.Println(array[1])
```

An array's length is static and can be accessed with the `len` function:

```
fmt.Println(len(array))
```

You can also duplicate an array using the `:=` operator and check array equality (that two arrays have the same elements) using the `==` operator.

```
duplicated := array
fmt.Println(array==duplicated)
duplicated[0]=0
fmt.Println(array==duplicated)
```

Slices

Slices are dynamic in size - they can grow and shrink as needed.

Slices are declared without a size:

```
var mySlice []int
```

definition

Slices are arrays

Slices are reference types that represent a segment of an underlying array.

Slices hold:
* a **reference** to the array data
* **length** - number of elements
* **capacity** - space in underlying array

Slices can be re-sliced to reference different parts of the array.

You can declare and instantiate an empty slice using the `make()` function:

```
mySlice := make([]int, 5)
fmt.Println(mySlice)
```

Or you can instantiate a slice from an existing array or from literals:

```
myArray := [5]int{1, 2, 3, 4, 5}
arraySlice := myArray[1:4]
fmt.Println(arraySlice)
numberSlice := []int{1, 2, 3}
fmt.Println(numberSlice)
```

Elements are accessed and modified using indexing, just like arrays. When slicing, the first number, indicating the starting index of the slice is inclusive, and the second number, indicating the stopping index of the slice is exclusive.

info

When to use Arrays vs Slices

- Use arrays when you know and need the exact number of elements. Arrays provide type safety, appendability, etc.
- Use slices when you need a dynamically sized sequence. Slices are more flexible and commonly used in Go.

In general, **slices will be the better choice for most use cases**. Arrays have some niche uses when the fixed size is required.

There are `len()` for length, and `cap()` for capacity functions. In addition, new elements can be added to a slice with `append()`:

```
moreNumbers := append(numberSlice, 4,5,6)
fmt.Println(moreNumbers)
```

The first parameter is the slice and all subsequent values are what is to be appended to the slice.

For Loop

There is only one loop in Go - the `for` loop.

The syntax of a `for` loop in Go is:

```
for init; condition; post {
  // loop body
}
```

- `init` - Executed before the first iteration. Often used to initialize variables.
- `condition` - Evaluated before each iteration. The loop executes as long as this is true.
- `post` - Executed at the end of each iteration. Often used to increment a counter.

Here is an example basic for loop that prints the numbers 0 through 9:

```
for i := 0; i < 10; i++ {
  fmt.Println(i)
}
```

This declares the variable `i`, checks it against the condition, and increments it each iteration.

`for` loops can act like `while` loops by omitting the init and post statements.

```
sum := 1
for sum < 1000 {
  sum += sum
}
fmt.Println(sum)
```

xdiscipline

Code Style: Automatic Code Formatting

There is a `go fmt` tool for automatic code formatting.

Try it out using `gofmt -w for.go`

To learn more about this command use `go help fmt`.

To create an infinite for loop, simply omit the loop condition. For example:

```
for {
  // loop forever
}
```

Infinite loops can be useful but should generally be avoided in favor of a loop with a terminating condition.

If you do need an infinite loop, make sure to include logic to `break` out of the loop manually.

Iterating over Arrays, Slices, and Strings

The `range` form of the for loop is used to iterate over arrays, slices, and strings.

When ranging over an array or slice, two values are returned on each iteration - the index and the value.

For example:

```
nums := []int{1, 2, 3}
for i, num := range nums {
  fmt.Println(i, num)
}
```

Strings return the index and **rune** on each iteration:

```
for i, c := range "hello" {
  fmt.Println(i, c)
}
```

xdiscipline

Runes

A **rune** literal is an integer that represents a Unicode code point:

```
'a' represents U+0061 which has integer value 97
'ä' represents U+00E4 which has integer value 228
```

Runes are written as character literals in single quotes:

```
'a' // rune for lowercase a
'ä' // rune for a-dieresis
```

A rune literal inside single quotes can be:
- A single printable character except `'` (e.g. `'a'`)
- A backslash escape sequence (e.g. `'\n'`)

Backslash escapes allow encoding Unicode values into ASCII text:
- + 2 hex digits (e.g. `'\x61'`)
- 4 hex digits (e.g. `'\u0061'`)
- 8 hex digits (e.g. `'\U00000061'`)
- 3 octal digits can also be used (e.g. `'\141'`)

Runes can be cast to strings as follows:

```go
var letter string = string('a')
fmt.Println(letter)
```

Nested For Loops

For loops can be nested to iterate over multidimensional data like nested arrays.

When nesting loops, make sure each variable has a unique name:

```go
for i := 0; i < 3; i++ {
  for j := 0; j < 5; j++ {
    fmt.Println(i, j)
  }
}
```

Use labels like `outer:` and `inner:` on nested loops to `break` or `continue` out of inner ones.

This prints the cartesian product of the iterators `i` and `j`.

Nested for loops are useful for tasks like iterating over 2D arrays:

```
var matrix [3][5]int // 3x5 2D array

for i := 0; i < 3; i++ {
  for j := 0; j < 5; j++ {
    fmt.Print(matrix[i][j])
  }
  fmt.Println() // newline after each row
}
```

If / Else and Switch / Case

Conditional statements are pieces of code that decide what the program will do next.

The conditional constructs in Go are If/Else and Switch/Case.

If / Else

The `if` condition takes an expression that returns a boolean value and executes the code in the subsequent curly braces when the boolean expression evaluates to true.

```
a := 7
if a > 5 {
  fmt.Println("Greater than 5")
}
```

An `else` branch can be added to specify behavior when the `if` condition is false.

```
a := 3
if a > 5 {
  fmt.Println("Greater than 5")
} else {
  fmt.Println("Less than or equal to 5")
}
```

You can chain `if {} else {}` structures together:

```
a := 2
if a == 1 {
    fmt.Println("One")
} else if a == 2 {
    fmt.Println("Two")
} else {
    fmt.Println("Three")
}
```

In the `if` construct, you can call a function that returns a boolean value and check its result:

```
if v := math.Pow(2, 0); v == 1 {
  fmt.Printf("All values raised to the value of 0 are 1")
} else {
  return v
}
```

Variables declared inside an if statement are also available inside any of the else blocks.

Switch / Case

The `switch` construct checks the value of some expression. The `case` statements define the values to be compared. If the value after the `case` statement matches the value of the expression from `switch`, then the code of the given `case` block is executed:

```
a := 3
switch a {
case 1:
    fmt.Println("one")
case 2:
    fmt.Println("two")
case 3, 4:
    fmt.Println("three or four")
default:
    fmt.Println("undefined")
}
```

`switch` may contain an optional `default` block, which is executed if none of the `case` statements contains the required value.

You can also specify multiple values after the `case` statement.

Additionally, `case` statements can support expressions:

```
num := 10
switch {
case num < 0:
  fmt.Println("Negative")
case num > 10:
  fmt.Println("Too high")
default:
  fmt.Println("Valid")
}
```

Functions

Functions in Go are defined using the `func` keyword. Arguments are passed by value, though slices, unlike arrays, are passed by reference.

In general, a function is declared as follows:

```
func FunctionName(parameters) returnTypes {
    //function body
}

//function call
FunctionCall(arguments)
```

For example, a function that does not take any parameters and does not return anything will look like:

```
func printHello() {
    fmt.Println("hello")
}

//call in main method
printHello()
```

In contrast, a function that takes parameters and returns a value will look like:

```
func add(a int, b int) int {
  return a + b
}

//call in main method
fmt.Println(add(3, 4))
```

Go functions can return more than one value. In this case, the return types are listed in parentheses:

```go
// divides two numbers and returns the quotient and remainder
func divide(x, y int) (int, int) {

  quotient := x / y
  remainder := x % y

  return quotient, remainder
}

//call in main method
quotient, remainder := divide(10, 3)
fmt.Println(quotient)
fmt.Println(remainder)
```

xdiscipline

Code Style: Function names

Like variables, functions are camelCase with the first letter indicating whether it is an internal function (lowercase) or an external-facing function (uppercase).

- Functions that return something are given noun-like names.
- Functions that do something are given verb-like names.

Function Parameter List

If several parameters have the same type, then the function parameters can be shortened by specifying the type after the group of variables:

```go
// before
func example(a int, b int, s1 string, s2 string) (int, bool) {
    ...
}

// after
func example(a, b int, s1, s2 string) (int, bool) {
    ...
}
```

In Go, a function can take an indefinite number of parameters – sometimes referred to as a **variadic function**. To do this, an ellipsis is placed before the type of these values.

```
func sum(numbers ...int) int {
    var result = 0
    for _, number := range numbers {
        result += number
    }
    return result
}
```

The call will look like this:

```
fmt.Println(sum(1, 2))
fmt.Println(sum(1, 2, 3, 4))
```

important

Ignoring variables _

The underscore _ in Go is used to ignore values when the variable is not needed.

In the sum function above, we use _ for the index variable in the for range loop. This is because we don't need the index, only the value.

By convention, using _ signals to the reader that the variable is intentionally ignored.

Passing Slices

Passing an indefinite number of arguments is not equivalent to passing a slice:

```go
func sum(numbers []int) int {
  total := 0
  for _, number := range numbers {
    total += number
  }
  return total
}

func main() {
  nums := []int{1, 2, 3, 4, 5}
  result := sum(nums)
  fmt.Println("Result:", result)
}
```

Error Handling

An error in Go is a return value with a built-in type `error`.

The `error` type is an interface type representing any value that can describe itself as a string:

```
type error interface {
    Error() string
}
```

Handling Errors

To handle an error in Go, you need to check the returned variable with type `error` from the function for `nil`.

```
user, err := getUser(id)
if err != nil {
    // log the error or perform other actions.
    logger.Debug(err.Error())
}
```

You can also ignore the error but it is not recommended:

```
user, _ := getUser(id)
```

Creating Custom Errors

The standard library has two built-in functions for creating errors: `errors.New` and `fmt.Errorf`. Both of these features allow us to specify a custom error message that you can display to your users.

`errors.New` takes one argument, the error message as a string.

The `fmt.Errorf` function formats according to the format specifier and returns an error.

```
package main

import (
    "errors"
    "fmt"
)

func main() {
    ...
}

func getUser(id) (User, error) {
    if (id == "") {
        return nil, errors.New("user id not defined")
        // or
        return nil, fmt.Errorf("user id not defined. current
time: %v", time.Now())
    }
}
```

Defer, Panic and Recover

When errors occur, Go stops the execution of the program and starts unwinding the call stack until it terminates the application or finds a function to handle the crash.

To deal with such errors, there is a “defer, panic, recover” mechanism.

Defer

`defer` pushes all function calls onto the application stack. In this case, the deferred functions are executed in the reverse order - regardless of whether a panic is caused or not. Defer can also be used to clean up resources, for example:

```
package main

import "fmt"

func main() {
    fmt.Println("start")
    defer deferFunc1()
    defer deferFunc2()
    fmt.Println("stop")
}

func deferFunc1() {
    fmt.Println("deferFunc1")
}

func deferFunc2() {
    fmt.Println("deferFunc2")
}
```

The output will be:
>

```
start
stop
deferFunc2
deferFunc1
```

As you can see from the output, the functions were called in reverse order.

Panic

`panic` tells us that the code cannot be executed further and stops the application from running. After the operator is called, all pending functions are executed and the program exits with a message about the cause of the panic and a stack trace.

Panic can also be invoked explicitly using the `panic()` method, which takes a string argument explaining the reason why we are panicking.

```
panic("panic")
```

Keep in mind that panicking is a bad practice when writing Go code.

Remember that when panicking, all pending functions will be executed (`defer`), and for example when calling `os.Exit()` they will not. Likewise, for example, when calling `log.Fatal()`. If you look at its implementation, you can see that it calls `os.Exit()`.

Recover

`recover()` function is needed to regain control in a panic. In this case, the operation of the application does not stop, but is restored and continues in normal mode.

Recover must always be called in the defer function. To report an error as a return value, you must call the recover function in the same goroutine as the panic, get the error structure from the recover function, and pass it to a variable.

```go
package main

import "fmt"

func main() {
    fmt.Println("start")
    funcWithRecover()
    fmt.Println("stop")
}

func funcWithRecover() {
    defer func() {
    if r := recover(); r != nil {
      fmt.Printf("Panic: %+v\n", r)
    }
  }()
    panic("panic in funcWithRecover")
}
```

The output will be:
>

```
start
Panic: panic in funcWithRecover
stop
```

As you can see, the program did not crash, but continued its execution. This can be useful, for example, if a function from some library may end in a panic, but at the same time we want to continue the execution of the program after handling it appropriately.