

Introduction to Golang

- Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.



Why should we learn a new Language?

- Go is expressive, concise, clean, and efficient – says a lot !!
- Go is easy and fun 😊
- Just play along and you'll find out



Installing Go

- The Go Playground, <http://play.golang.org>
- The Go Playground is a web service that runs on golang.org's servers. The service receives a Go program, compiles, links, and runs the program inside a sandbox, then returns the output.

Lecture 05 - Introduction to Golang

<https://golang.org>

The Go Programming Language

[Documents](#)

[Packages](#)

[The Project](#)


[Help](#)

[Blog](#)

Try Go

Pop-out 

```
// You can edit this code!  
// Click here and start typing.  
package main  
  
import "fmt"  
  
func main() {  
    fmt.Println("Hello, 世界")  
}
```

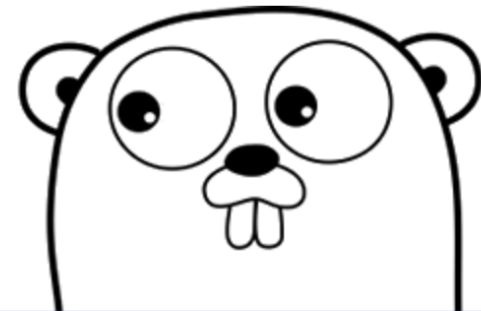
Hello, World! 

Run

Share

Tour

Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.



Download Go

Binary distributions available for
Linux, Mac OS X, Windows, and more.

Installing Go

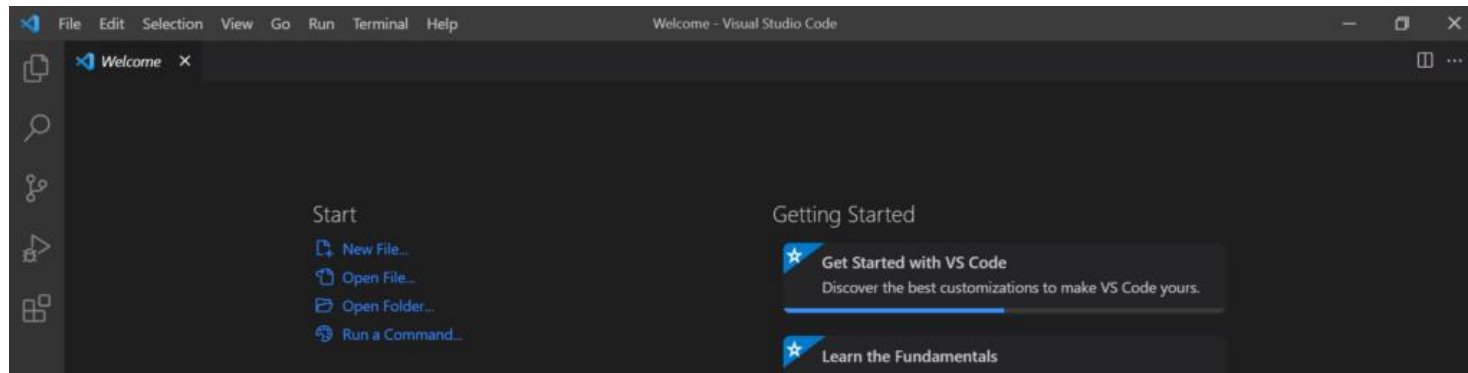
- You should however install it locally
 - Follow instructions at <https://golang.org/doc/install>, simple steps and binaries provided.
 - Download and install using the setup and do the following to test your work
 - The GOPATH environment variable should be setup

Some editors with support for Go

- Visual studio code
- SublimeText with GoSublime
- Atom with go-plus

Getting started (Install Visual Studio Code)

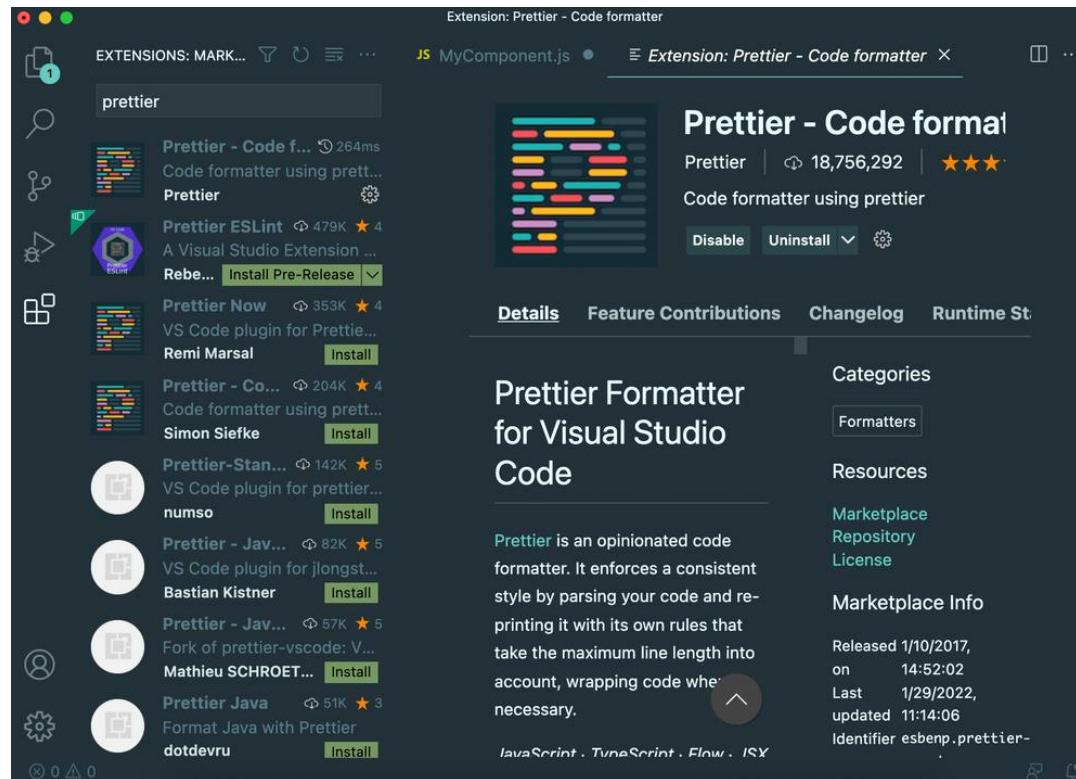
- Visual Studio Code is the free and open-sourced code editor. It is one of the top most editor used especially for JavaScript application development.
- You can [download Visual Studio Code](#) according to your OS and system requirements. After download, install it. After opening, it will look like this.



Getting started

(Format your code with the Prettier extension)

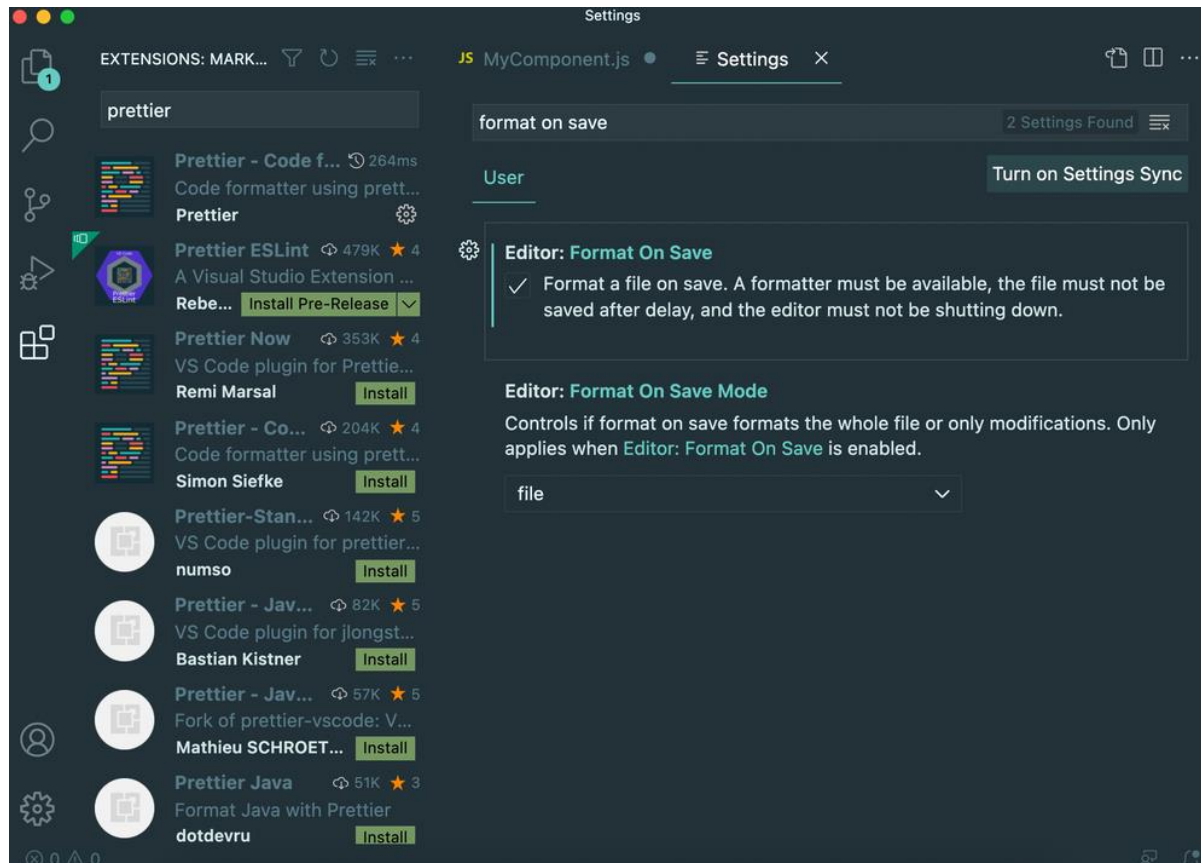
- To be able to instantly format our code every time we save a .js file, we can again go to the extensions tab (⇧ + ⌘ (Ctrl) + X), type in "prettier" and install it:



Getting started

(Format your code with the Prettier extension)

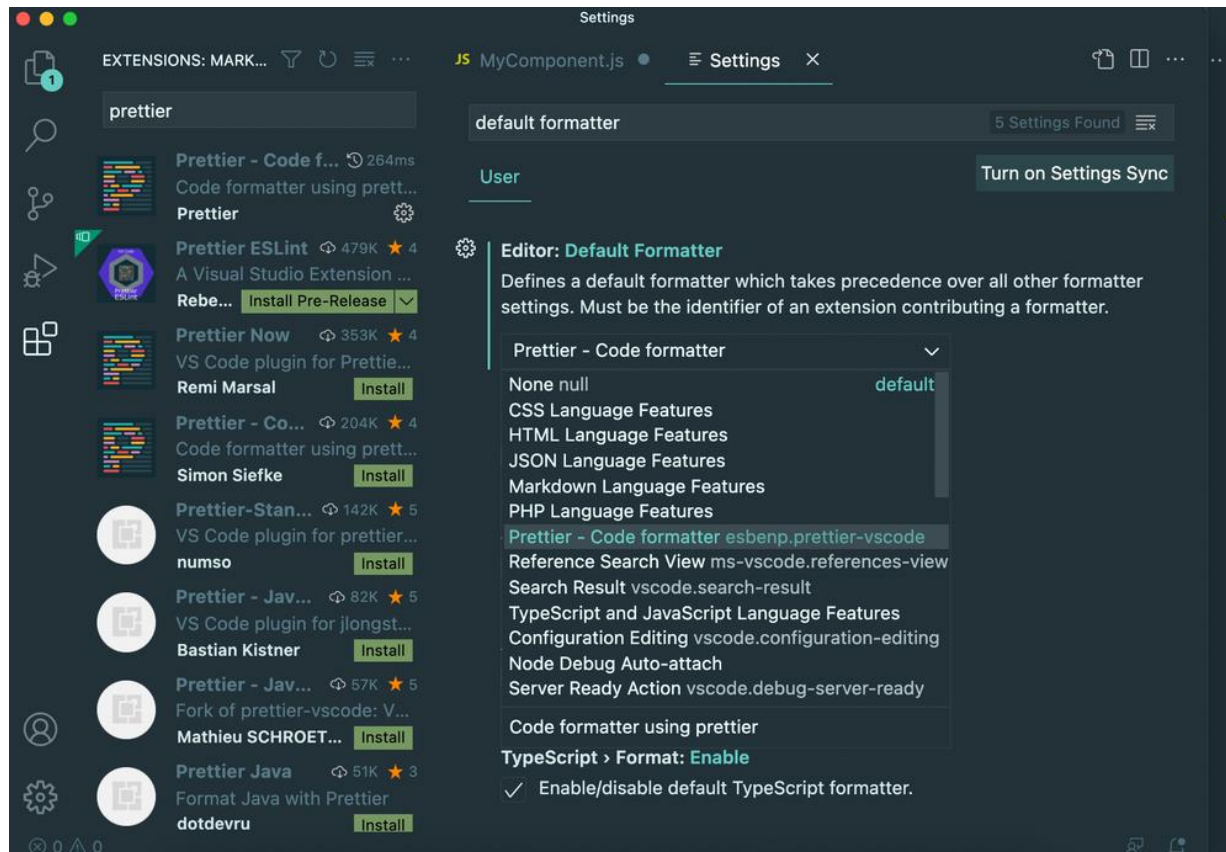
- Next, we can go back to preferences -> setting, and search for "**format on save**" and make sure it is checked:



Getting started

(Format your code with the Prettier extension)

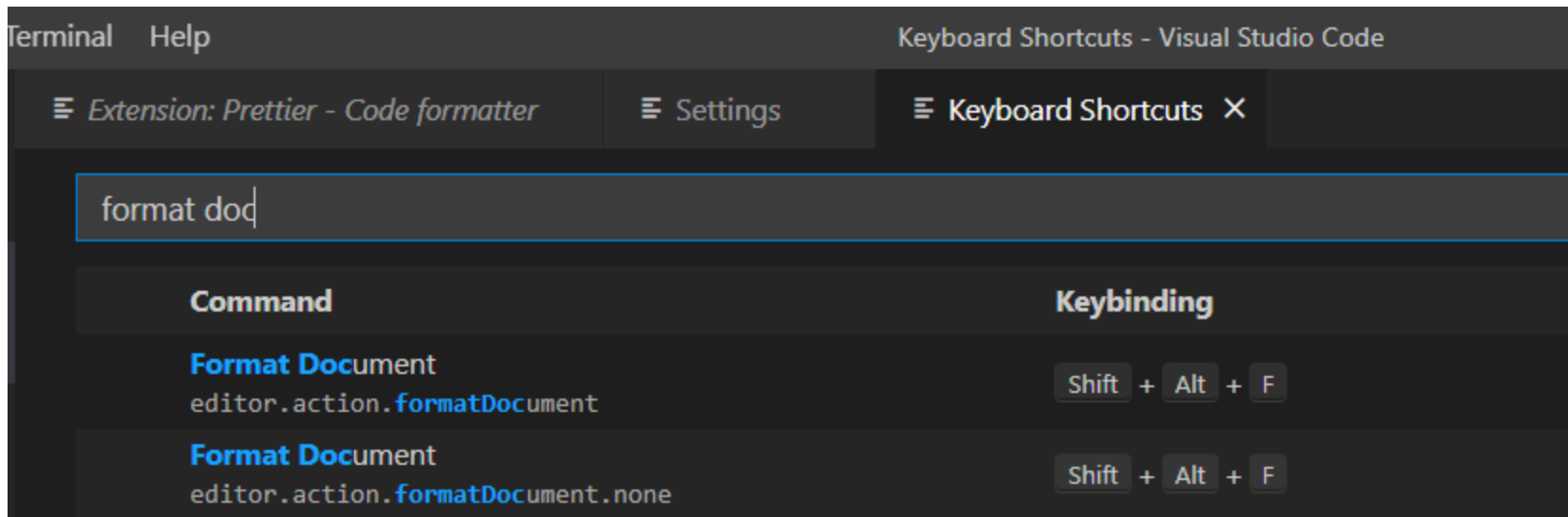
- Next, we can go back to preferences -> setting, and search for the "**default formatter**" setting and set it to Prettier



Getting started

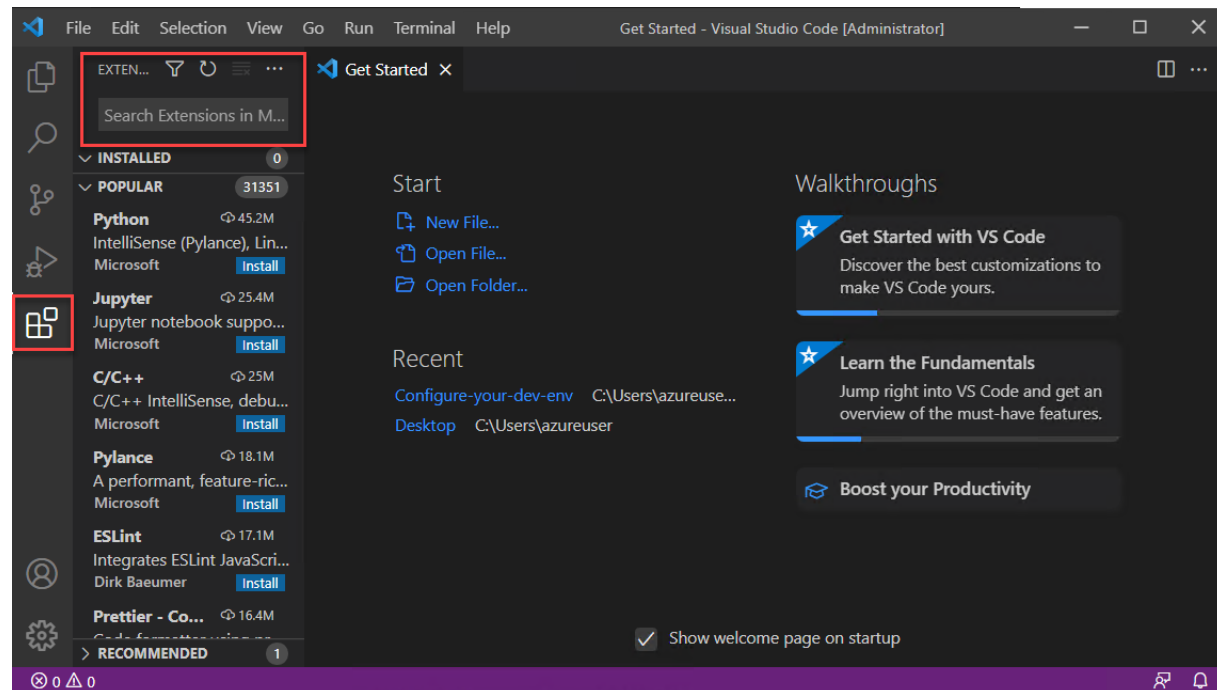
(Format your code with the Prettier extension)

- Next, we can go back to preferences -> Keyboard shortcuts and look for the shortcut to use for formatting code.



Installing go!

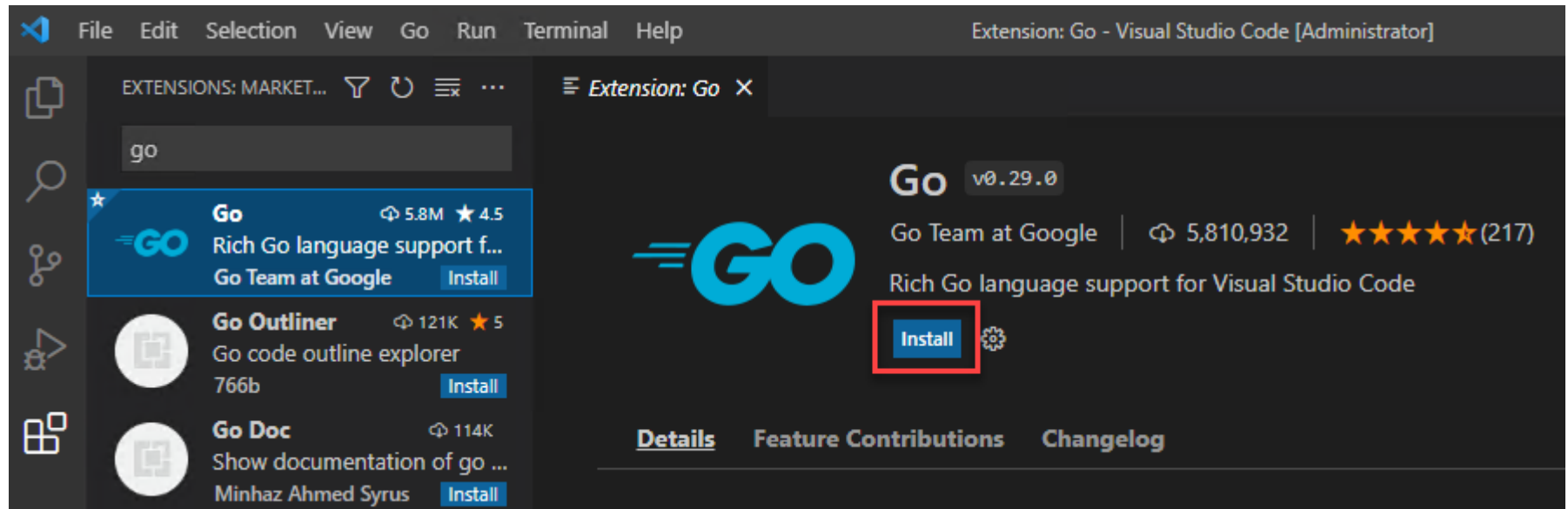
- In Visual Studio Code, bring up the Extensions view by clicking on the Extensions icon in the Activity Bar. Or use keyboard shortcut (Ctrl+Shift+X).



Lecture 05 - Introduction to Golang

Installing go!

- Search for the Go extension, then select install.



Hello World !!

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    fmt.Printf("Hello, world.\n")
```

```
}
```


Let's not rush

- Before we can move on, it is better to get organized.
- Good organization of our code and directories would do wonders 😊

Go and GitHub

- The go tool is designed to work with open source code maintained in public repositories.
- Although you don't need to publish your code, the model for how the environment is set up works the same whether you do or not.

Workspaces

- Go code must be kept inside a workspace.
- A workspace is a directory hierarchy with three directories at its root:
 - src contains Go source files organized into packages (one package per directory),
 - pkg contains package objects, and
 - bin contains executable commands.
- The go tool builds source packages and installs the resulting binaries to the pkg and bin directories.

An example

```
bin/
  hello                # command executable
  outyet               # command executable
pkg/
  linux_amd64/
    github.com/golang/example/
      stringutil.a      # package object
src/
  github.com/golang/example/
    .git/               # Git repository metadata
    hello/
      hello.go          # command source
    outyet/
      main.go           # command source
      main_test.go      # test source
    stringutil/
      reverse.go        # package source
      reverse_test.go   # test source
```

Get ready to use Git!

(Create a Git repository for all your tasks)

1. Set the name and email for Git to use when you commit:

```
$ git config --global user.name "Bugs Bunny"
```

```
$ git config --global user.email bugs@gmail.com
```

- You can call `git config -list` to verify these are set.
- These will be set globally for all Git projects you work with.
- You can also set variables on a project-only basis by not using the `--global` flag.

A Tour of Go

<http://play.golang.org>

Hello World !!

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    fmt.Printf("Hello, world.\n")
```

```
}
```

Printing on Console - The fmt package

```
package main

import "fmt"

func main() {
    // The Println method can handle one or more arguments.
    fmt.Println("cat")
    fmt.Println("cat", 900)

    // Use Println on a slice.
    items := []int{10, 20, 30}
    fmt.Println(items)
}
```

Output

```
cat
cat 900
[10 20 30]
```


Printing on Console - The fmt package

```
package main

import "fmt"

func main() {
    name := "Mittens"
    weight := 12

    // Use %s to mean string.
    // ... Use an explicit newline.
    fmt.Printf("The cat is named %s.\n", name)
    // Use %d to mean integer.
    fmt.Printf("Its weight is %d.\n", weight)
}
```

Output

```
The cat is named Mittens.
Its weight is 12.
```

Printing on Console - The fmt package

```
package main

import "fmt"

func main() {
    elements := []int{999, 99, 9}

    // Loop over the int slice and Print its elements.
    // ... No newline is inserted after Print.
    for i := 0; i < len(elements); i++ {
        fmt.Print(elements[i] + 1)
        fmt.Print(" ")
    }
    fmt.Println("... DONE!")
}
```

Output

```
1000 100 10 ... DONE!
```

Printing on Console - The fmt package

```
package main

import "fmt"

func main() {
    result := true
    name := "Spark"
    size := 2000
    // Print line with v format codes.
    fmt.Printf("Result = %v, Name = %v, Size = %v",
        result, name, size)
}
```

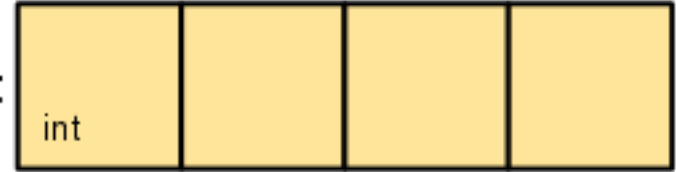
Output

```
Result = true, Name = Spark, Size = 2000
```

Arrays

Arrays

[4]int

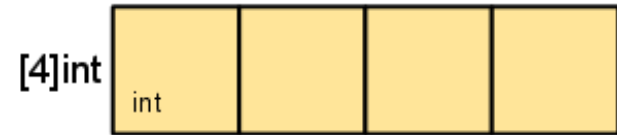


- Go's arrays are values.
- An array variable denotes the entire array; it is not a pointer to the first array element (as in C/C++).
- This means that when you assign or pass around an array value you will make a copy of its contents.

```
var array_name := [length]datatype{values} // here length is defined  
or
```

```
var array_name := [...]datatype{values} // here length is inferred
```

Arrays



This example declares two arrays [arr1(inferred) and arr2(defined length)]

```
package main
import ("fmt")

func main() {
    var arr1 = [...]int{1, 2, 3}
    //creates a variable of a particular type, attaches a name to it, and sets its initial value

    arr2 := [5]int{4, 5, 6, 7, 8} // := is called short variable declaration which takes form
    var arr3 = [4]string{"Volvo", "BMW", "Ford", "Mazda"}

    fmt.Println(arr1)
    fmt.Println(arr2)
    fmt.Println(arr3)
    fmt.Println(arr1[0])
    fmt.Println(arr2[2])
    fmt.Println(len(arr2))
}
```

Passing Arrays

- Go's documentation makes it clear that arrays are passed by copy.
- How to achieve pass by reference?
 - *Option A - use pointers, complex and less elegant*
 - *Option B - Use slices*

Passing Arrays

- Go's documentation makes it clear that arrays are passed by copy.

```
package main

import "fmt"

func main() {

    strArray1 := [3]string{"Japan", "Australia", "Germany"}
    strArray2 := strArray1 // data is passed by value
    strArray3 := &strArray1 // data is passed by reference

    fmt.Printf("strArray1: %v\n", strArray1)
    fmt.Printf("strArray2: %v\n", strArray2)

    strArray1[0] = "Canada"

    fmt.Printf("strArray1: %v\n", strArray1)
    fmt.Printf("strArray2: %v\n", strArray2)
    fmt.Printf("*strArray3: %v\n", *strArray3)
}
```

Output

```
strArray1: [Japan Australia Germany]
strArray2: [Japan Australia Germany]
strArray1: [Canada Australia Germany]
strArray2: [Japan Australia Germany]
*strArray3: [Canada Australia Germany]
```


Go Struct

Go Struct

While arrays are used to store multiple values of the same data type into a single variable, **structs** are used to store multiple values of different data types into a single variable.

Syntax

```
type struct_name struct {  
    member1 datatype;  
    member2 datatype;  
    member3 datatype;  
    ...  
}
```

Example

Here we declare a struct type `Person` with the following members: `name`, `age`, `job` and `salary`:

```
type Person struct {  
    name string  
    age int  
    job string  
    salary int  
}
```

Go Struct (Access Struct Members)

To access any member of a structure, use the dot operator (.) between the structure variable name and the structure member:

```
package main
import ("fmt")

type Person struct {
    name string
    age  int
    job  string
    salary int
}

func main() {
    var pers1 Person
    var pers2 Person

    // Pers1 specification
    pers1.name = "Hege"
    pers1.age = 45
    pers1.job = "Teacher"
    pers1.salary = 6000

    // Pers2 specification
    pers2.name = "Cecilie"
    pers2.age = 24
    pers2.job = "Marketing"
    pers2.salary = 4500

    // Access and print Pers1 info
    fmt.Println("Name: ", pers1.name)
    fmt.Println("Age: ", pers1.age)
    fmt.Println("Job: ", pers1.job)
    fmt.Println("Salary: ", pers1.salary)

    // Access and print Pers2 info
    fmt.Println("Name: ", pers2.name)
    fmt.Println("Age: ", pers2.age)
    fmt.Println("Job: ", pers2.job)
    fmt.Println("Salary: ", pers2.salary)
}
```

Task 1 Pass Struct as Function Arguments

- Using the previous example, demonstrate how to pass a structure as a function argument and print its values.

- Function syntax

```
func FunctionName() {  
    // code to be executed  
}
```

Struct with an Array

```
package main

import "fmt"

type Doctor struct {
    number      int
    doctorName  string
    patients    []string
}

func main() {
    aDoctor := Doctor{
        number:      006,
        doctorName:  "Naveed",
        patients:    []string{"A", "B", "C", "D"},
    }
    fmt.Printf("Doctor ID=%v, Doctor name=%v, Patients =%v", aDoctor.number, aDoctor.doctorName, aDoctor.patients)
}
```

Task 2 array of structs

- Here, you can see the two different structs. The company struct uses the employee struct as an array. Write a program that do the following:
 - Add three employee using the employee struct (e.g., emp1 := employee{"Amir", 80000, "Full-Stack Developer"})
 - Create an array of 'empls' by add the above three records (empls:=[]employee{ ...})
 - create a company struct and add values to it (e.g., {"Tetra", empls})
 - Print company details

```
type employee struct {  
    name      string  
    salary    int  
    position  string  
}  
  
type company struct {  
    companyName string  
    employees   []employee  
}
```

Methods with struct

- Go methods are similar to Go function with one difference, i.e, the method contains a receiver argument in it.
- With the help of the receiver argument, the method can access the properties of the receiver.

```
func(receiver_name Type)method_name(parameter_list)(return_type)
{
// Code
}
```

Methods without struct

```
import (  
    "fmt"  
)  
  
type StudentRecord struct {  
    rollnumber int  
    name       string  
    address    string  
}  
  
func AddStudent(rollno int, name string, address string) *StudentRecord {  
    s := new(StudentRecord)  
    s.rollnumber = rollno  
    s.name = name  
    s.address = address  
    return s  
}  
  
func main() {  
  
    st := AddStudent(24, "Ehtesham", "aaaaaaaaaaaaa")  
    fmt.Println(&st, st)  
    st = AddStudent(25, "Naveed", "bbbbbbbbbbbbbbb")  
    fmt.Println(&st, st)  
}
```


Methods with struct

```
package main

import (
    "fmt"
)

type StudentRecord struct {
    rollnumber int
    name       string
    address    string
}

func (s *StudentRecord) AddStudent(rollno int, name string, address string) *StudentRecord {
    s.rollnumber = rollno
    s.name = name
    s.address = address
    return s
}

func main() {
    st := new(StudentRecord)
    st.AddStudent(24, "Asim", "Bahria Town")
    fmt.Println(&st, st)
}
```

Methods with struct - Creating a list of students

```
import (  
    "fmt"  
    "strings"  
)  
  
type Student struct {  
    rollnumber int  
    name       string  
    address    string  
}  
  
func NewStudent(rollno int, name string, address string)  
*Student {  
    s := new(Student)  
    s.rollnumber = rollno  
    s.name = name  
    s.address = address  
    return s  
}  
  
type StudentList struct {  
    list []*Student  
}  
  
func (ls *StudentList) CreateStudent(rollno int, name  
string, address string) *Student {  
    st := NewStudent(rollno, name, address)  
    ls.list = append(ls.list, st)  
    return st  
}  
  
func main() {  
    student := new(StudentList)  
    student.CreateStudent(24, "Asim", "AAAAAA")  
    student.CreateStudent(25, "Naveed", "BBBBBB")  
    fmt.Println(student)  
}
```

Task 3- Creating a list of students

- Currently, the program output the addresses. Your task is to write a Print() method(s) to display the students data from the student list in the format given below
- The strings.Repeat can help you print the '=' for the specified number of time

```
no :=1
```

```
fmt.Printf("%s List %d %s\n", strings.Repeat("=", 25), no, strings.Repeat("=", 25))
```

```
===== List 0 =====  
student rollno      24  
student name        Asim  
student address      AAAAAA  
===== List 1 =====  
student rollno      25  
student name        Naveed  
student address      BBBBBB
```

Calculating Hash

- Package sha256 implements the SHA224 and SHA256 hash algorithms as defined in FIPS 180-4

<https://pkg.go.dev/crypto/sha256#example-Sum256>

```
package main

import (
    "crypto/sha256"
    "fmt"
)

func main() {
    sum := sha256.Sum256([]byte("hello world\n"))
    fmt.Printf("%x\n", sum) //hexadecimal
}
```

Calculating Hash

```
package main
import (
    "crypto/sha256" //package sha256 implements the SHA224 and SHA256 hash algorithms
    "fmt"
)

func CalculateHash (stringToHash string) string { //function for calculating hash of a block,
stringToHash is a string containing data for which hash is to be calculated (passed as function
argument while calling this function in main function), the return type of this function is string
i.e., this function will return the hash

    fmt.Printf("String Received: %s\n", stringToHash) //%s is to print a string, \n is
for new line, Printf stands for Print Formatter function in fmt package. It prints formatted
strings.
    return fmt.Sprintf("%x", sha256.Sum256([]byte(stringToHash))) //hash calculation through sha256
hash function, []byte is a small header pointing to data with lengths indicating how much data is
present. []byte has two lengths: the current length of the data and the capacity. The capacity tells
us how many more bytes we can add to the data before needing to go and get a bigger piece of memory.
%x represents base 16 or hexadecimal value for hash, Sprintf is for printing formatted output and it
returns the resulted string

}

func main() { //main function

    output := CalculateHash("AliceToBob")
    fmt.Printf("Hash: %x\n", output)
}
```

Task 4- Creating a list of students

- Extending task 3, add an array for the subject a student is currently studying.
- Calculate hash of the block data and display it